「卡码网」23种设计模式精讲

卡码网设计模式精讲，已经开源到github上：https://github.com/youngyangyang04/kama-DesignPattern ，**支持Java，C++，Python，Go多个版本**，欢迎star支持一波，也可以提交pr，贡献其他语言版本。

本PDF为Java讲解，其他语言版本 则开源的 Github上，大家可以自己去学习。

设计模式专属练习题库：https://kamacoder.com/designpattern.php

前言

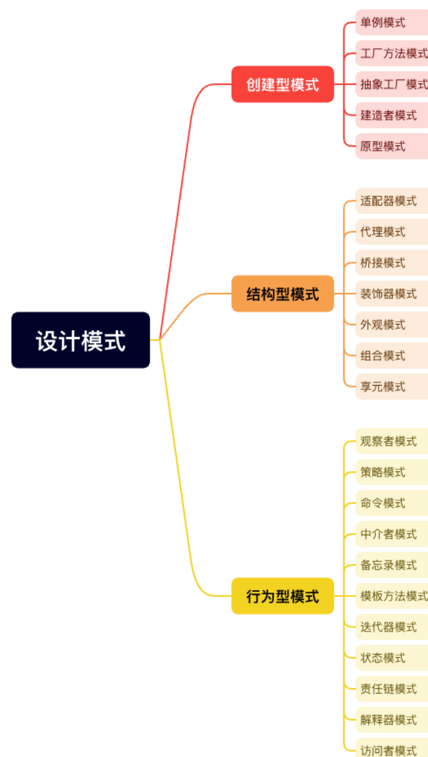
“设计模式”虽然被公认为是软件开发的最佳实践，但也一直是“高深、玄妙”的代名词，这也导致很多人对其望而生畏，缺乏对其概念的基本认识。

因此卡码网推出了设计模式基础教程，**免费公开给本站的用户**，旨在让你对23种“设计模式”有所了解，能知道其基本骨架，初步认识设计模式的使用场景。

区别于网上其他教程，本教程的特点是:

- **23种设计模式全覆盖**，涵盖了所有Gang of Four设计模式，包括创建型、结构型和行为型设计模式。
- 通过23道简单而实用的例子，**以刷算法题的形式了解每种设计模式的概念、结构和应用场景**。
- **为每个设计模式提供清晰的文字解释、结构图和代码演示**，帮助你更好地理解和实践。
- **难度安排循序渐进**，从基础的、常用的设计模式逐步深入。

设计模式大纲：



本PDF提供 23种设计模式的讲解，同时给出对应的练习题目，在：设计模式专属练习题库

单例模式

题目链接

[单例模式-小明的购物车]

什么是单例设计模式

单例模式是一种**创建型设计模式**， 它的核心思想是保证**一个类只有一个实例，并提供一个全局访问点来访问这个实例。**

- 只有一个实例的意思是，在整个应用程序中，只存在该类的一个实例对象，而不是创建多个相同类型的对象。
- 全局访问点的意思是，为了让其他类能够获取到这个唯一实例，该类提供了一个全局访问点（通常是一个静态方法），通过这个方法就能获得实例。

为什么要使用单例设计模式呢

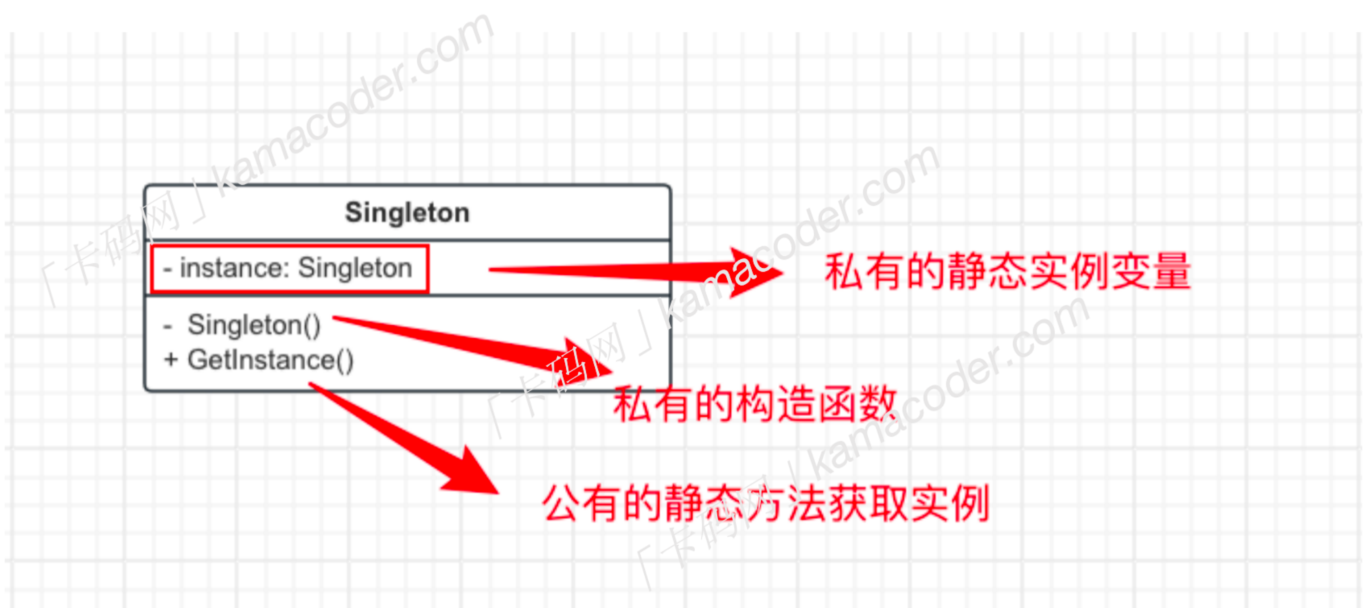
简易来说，单例设计模式有以下几个优点让我们考虑使用它：

- 全局控制：保证只有一个实例，这样就可以严格的控制客户怎样访问它以及何时访问它，简单的说就是对唯一实例的受控访问（引用自《大话设计模式》第21章）
- 节省资源：也正是因为只有一个实例存在，就避免多次创建了相同的对象，从而节省了系统资源，而且多个模块还可以通过单例实例共享数据。
- 懒加载：单例模式可以实现懒加载，只有在需要时才进行实例化，这无疑会提高程序的性能。

单例设计模式的基本要求

想要实现一个单例设计模式，必须遵循以下规则：

- 私有的构造函数：防止外部代码直接创建类的实例
- 私有的静态实例变量：保存该类的唯一实例
- 公有的静态方法：通过公有的静态方法来获取类的实例



单例设计模式的实现

单例模式的实现方式有多种，包括懒汉式、饿汉式等。

饿汉式指的是在**类加载时就已经完成了实例的创建**，不管后面创建的实例有没有使用，先创建再说，所以叫做“饿汉”。

而懒汉式指的是只有在请求实例时才会创建，如果在首次请求时还没有创建，就创建一个新的实例，如果已经创建，就返回已有的实例，意思就是**需要使用了再创建**，所以称为“懒汉”。

在多线程环境下，由于饿汉式在程序启动阶段就完成了实例的初始化，因此不存在多个线程同时尝试初始化实例的问题，但是懒汉式中多个线程同时访问 `getInstance()` 方法，并且在同一时刻检测到实例没有被创建，就可能会同时创建实例，从而导致多个实例被创建，这种情况下我们可以采用一些同步机制，例如使用互斥锁来确保在任何时刻只有一个线程能够执行实例的创建。

> 举个例子，你和小明都发现家里没米了，在你们没有相互通知的情况下，都会去超市买一袋米，这样就重复购买了，违背了单例模式。

下面以Java的代码作为实例，说明单例设计模式的基本写法：

1. 饿汉模式：实例在类加载时就被创建, 这种方式的实现相对简单，但是实例有可能没有使用而造成资源浪费。

```java
public class Singleton {
    private static final Singleton instance = new Singleton();

    private Singleton() {
        // 私有构造方法，防止外部实例化
    }

    public static Singleton getInstance() {
        return instance;
    }
}
```

2. 懒汉模式：第一次使用时才创建

```java
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        // 私有构造方法，防止外部实例化
    }
    // 使用了同步关键字来确保线程安全, 可能会影响性能
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

在懒汉模式的基础上，可以使用双重检查锁来提高性能。

```
public class Singleton {
    private static volatile Singleton instance;

    private Singleton() {
        // 私有构造方法，防止外部实例化
    }

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

什么时候使用单例设计模式

说了这么多，那在什么场景下应该考虑使用单例设计模式呢？可以结合单例设计模式的优点来看。

1. 资源共享

多个模块共享某个资源的时候，可以使用单例模式，比如说应用程序需要一个全局的配置管理器来存储和管理配置信息、亦或是使用单例模式管理数据库连接池。

2. 只有一个实例

当系统中某个类只需要一个实例来协调行为的时候，可以考虑使用单例模式， 比如说管理应用程序中的缓存，确保只有一个缓存实例，避免重复的缓存创建和管理，或者使用单例模式来创建和管理线程池。

3. 懒加载

如果对象创建本身就比较消耗资源，而且可能在整个程序中都不一定会使用，可以使用单例模式实现懒加载。

在许多流行的工具和库中，也都使用到了单例设计模式，比如Java中的`Runtime`类就是一个经典的单例，表示程序的运行时环境。此外 Spring 框架中的应用上下文 (`ApplicationContext`) 也被设计为单例，以提供对应用程序中所有 bean 的集中式访问点。

本题代码

```
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Scanner;

public class Main {

```

```java
    public static void main(String[] args) {
        ShoppingCartManager cart = ShoppingCartManager.getInstance();
        Scanner scanner = new Scanner(System.in);

        while (scanner.hasNext()) {
            String itemName = scanner.next();
            int quantity = scanner.nextInt();

            // 获取购物车实例并添加商品
            cart.addToCart(itemName, quantity);
        }

        // 输出购物车内容
        cart.viewCart();
    }
}

class ShoppingCartManager {

    // 饿汉模式实现单例
    private static final ShoppingCartManager instance = new ShoppingCartManager();

    // 购物车存储商品和数量的映射
    private Map<String, Integer> cart;

    // 私有构造函数
    private ShoppingCartManager() {
        cart = new LinkedHashMap<>();
    }

    // 获取购物车实例
    public static ShoppingCartManager getInstance() {
        return instance;
    }

    // 添加商品到购物车
    public void addToCart(String itemName, int quantity) {
        cart.put(itemName, cart.getOrDefault(itemName, 0) + quantity);
    }

    // 查看购物车
    public void viewCart() {
        for (Map.Entry<String, Integer> entry : cart.entrySet()) {
            System.out.println(entry.getKey() + " " + entry.getValue());
        }
    }
}
```

工厂方法模式

题目链接

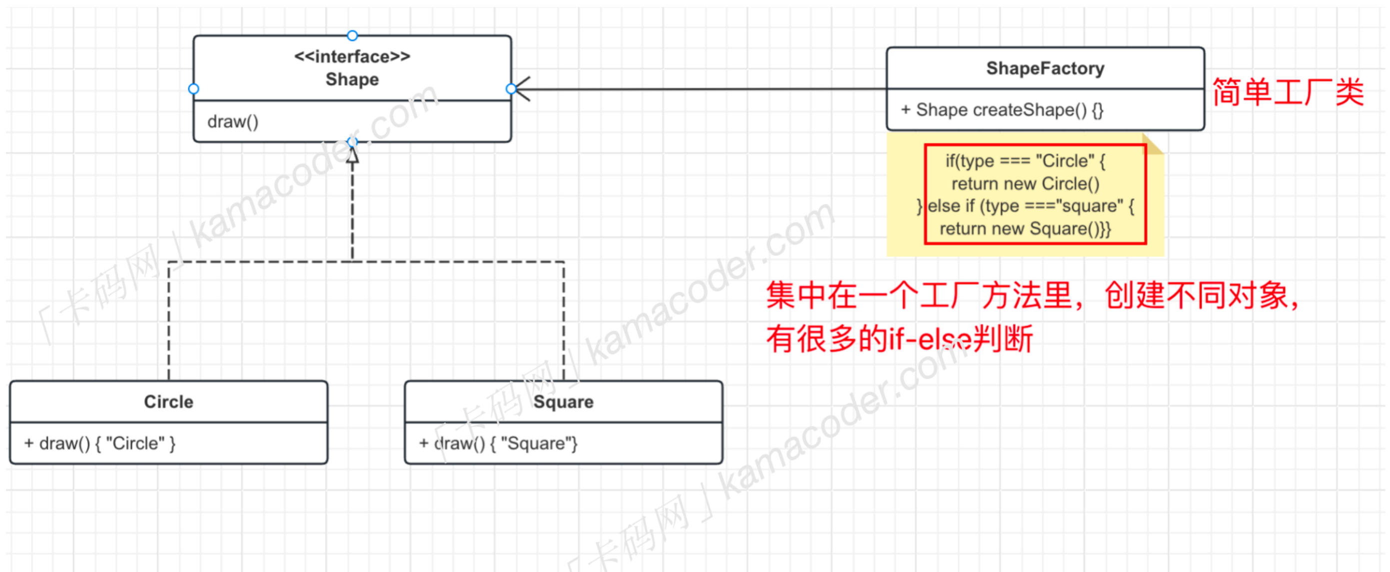
[工厂方法模式-积木工厂]

简单工厂模式

在了解工厂方法模式之前，有必要对"简单工厂"模式进行一定的了解，简单工厂模式是一种创建型设计模式，但并不属于23种设计模式之一，更多的是一种编程习惯。

简单工厂模式的核心思想是将产品的创建过程封装在一个工厂类中，把创建对象的流程集中在这个工厂类里面。

简单工厂模式包括三个主要角色，工厂类、抽象产品、具体产品，下面的图示则展示了工厂类的基本结构。



- 抽象产品，比如上图中的`Shape`接口，描述产品的通用行为。
- 具体产品：实现抽象产品接口或继承抽象产品类，比如上面的`Circle`类和`Square`类，具体产品通过简单工厂类的`if-else`逻辑来实例化。
- 工厂类：负责创建产品，根据传递的不同参数创建不同的产品示例。

简单工厂类简化了客户端操作，客户端可以调用工厂方法来获取具体产品，而无需直接与具体产品类交互，降低了耦合，但是有一个很大的问题就是不够灵活，如果需要添加新的产品，就需要修改工厂类的代码。

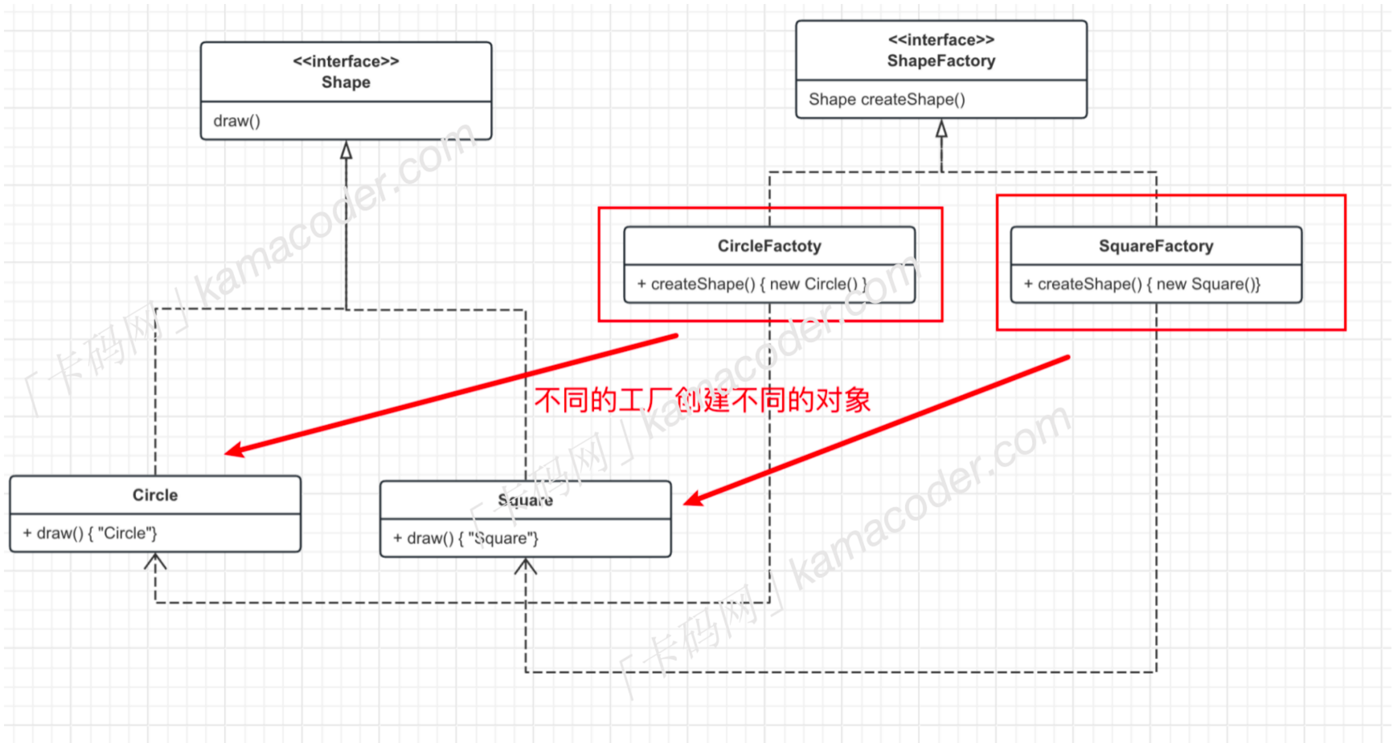
什么是工厂方法模式

工厂方法模式也是一种创建型设计模式，简单工厂模式只有一个工厂类，负责创建所有产品，如果要添加新的产品，通常需要修改工厂类的代码。而工厂方法模式引入了抽象工厂和具体工厂的概念，每个具体工厂只负责创建一个具体产品，添加新的产品只需要添加新的工厂类而无需修改原来的代码，这样就使得产品的生产更加灵活，支持扩展，符合开闭原则。

工厂方法模式分为以下几个角色：

- 抽象工厂：一个接口，包含一个抽象的工厂方法（用于创建产品对象）。
- 具体工厂：实现抽象工厂接口，创建具体的产品。
- 抽象产品：定义产品的接口。
- 具体产品：实现抽象产品接口，是工厂创建的对象。

> 实际上工厂方法模式也很好理解，就拿"手机Phone"这个产品举例，手机是一个抽象产品，小米手机、华为手机、苹果手机是具体的产品实现，而不同品牌的手机在各自的生产厂家生产。



基本实现

根据上面的类图，我们可以写出工厂方法模式的基本实现。

```java
// 抽象产品
interface Shape {
    void draw();
}

// 具体产品 - 圆形
class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Circle");
    }
}

// 具体产品 - 正方形
class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Square");
    }
}

// 抽象工厂
interface ShapeFactory {
    Shape createShape();
}
```

```java
// 具体工厂 - 创建圆形
class CircleFactory implements ShapeFactory {
    @Override
    public Shape createShape() {
        return new Circle();
    }
}

// 具体工厂 - 创建正方形
class SquareFactory implements ShapeFactory {
    @Override
    public Shape createShape() {
        return new Square();
    }
}

// 客户端代码
public class Client {
    public static void main(String[] args) {
        ShapeFactory circleFactory = new CircleFactory();
        Shape circle = circleFactory.createShape();
        circle.draw();  // 输出：Circle

        ShapeFactory squareFactory = new SquareFactory();
        Shape square = squareFactory.createShape();
        square.draw();  // 输出：Square
    }
}
```

应用场景

工厂方法模式使得每个工厂类的职责单一，每个工厂只负责创建一种产品，当创建对象涉及一系列复杂的初始化逻辑，而这些逻辑在不同的子类中可能有所不同时，可以使用工厂方法模式将这些初始化逻辑封装在子类的工厂中。在现有的工具、库中，工厂方法模式也有广泛的应用，比如：

- Spring 框架中的 Bean 工厂：通过配置文件或注解，Spring 可以根据配置信息动态地创建和管理对象。
- JDBC 中的 Connection 工厂：在 Java 数据库连接中，`DriverManager` 使用工厂方法模式来创建数据库连接。不同的数据库驱动（如 MySQL、PostgreSQL 等）都有对应的工厂来创建连接。

本题代码

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

// 抽象积木接口
interface Block {
    void produce();
}
```

```java
// 具体圆形积木实现
class CircleBlock implements Block {
    @Override
    public void produce() {
        System.out.println("Circle Block");
    }
}

// 具体方形积木实现
class SquareBlock implements Block {
    @Override
    public void produce() {
        System.out.println("Square Block");
    }
}

// 抽象积木工厂接口
interface BlockFactory {
    Block createBlock();
}

// 具体圆形积木工厂实现
class CircleBlockFactory implements BlockFactory {
    @Override
    public Block createBlock() {
        return new CircleBlock();
    }
}

// 具体方形积木工厂实现
class SquareBlockFactory implements BlockFactory {
    @Override
    public Block createBlock() {
        return new SquareBlock();
    }
}

// 积木工厂系统
class BlockFactorySystem {
    private List<Block> blocks = new ArrayList<>();

    public void produceBlocks(BlockFactory factory, int quantity) {
        for (int i = 0; i < quantity; i++) {
            Block block = factory.createBlock();
            blocks.add(block);
            block.produce();
        }
    }
```

```
    public List<Block> getBlocks() {
        return blocks;
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 创建积木工厂系统
        BlockFactorySystem factorySystem = new BlockFactorySystem();

        // 读取生产次数
        int productionCount = scanner.nextInt();
        scanner.nextLine();

        // 读取每次生产的积木类型和数量
        for (int i = 0; i < productionCount; i++) {
            String[] productionInfo = scanner.nextLine().split(" ");
            String blockType = productionInfo[0];
            int quantity = Integer.parseInt(productionInfo[1]);

            if (blockType.equals("Circle")) {
                factorySystem.produceBlocks(new CircleBlockFactory(), quantity);
            } else if (blockType.equals("Square")) {
                factorySystem.produceBlocks(new SquareBlockFactory(), quantity);
            }
        }
    }
}
```

抽象工厂模式

题目链接

[抽象工厂模式-家具工厂]

什么是抽象工厂模式

抽象工厂模式也是一种创建型设计模式，提供了一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类【引用自大话设计模式第15章】

这样的描述似乎理解起来很困难，我们可以把它与【工厂方法模式】联系起来看。

之前我们已经介绍了“工厂方法模式”，那为什么还有要抽象工厂模式呢?

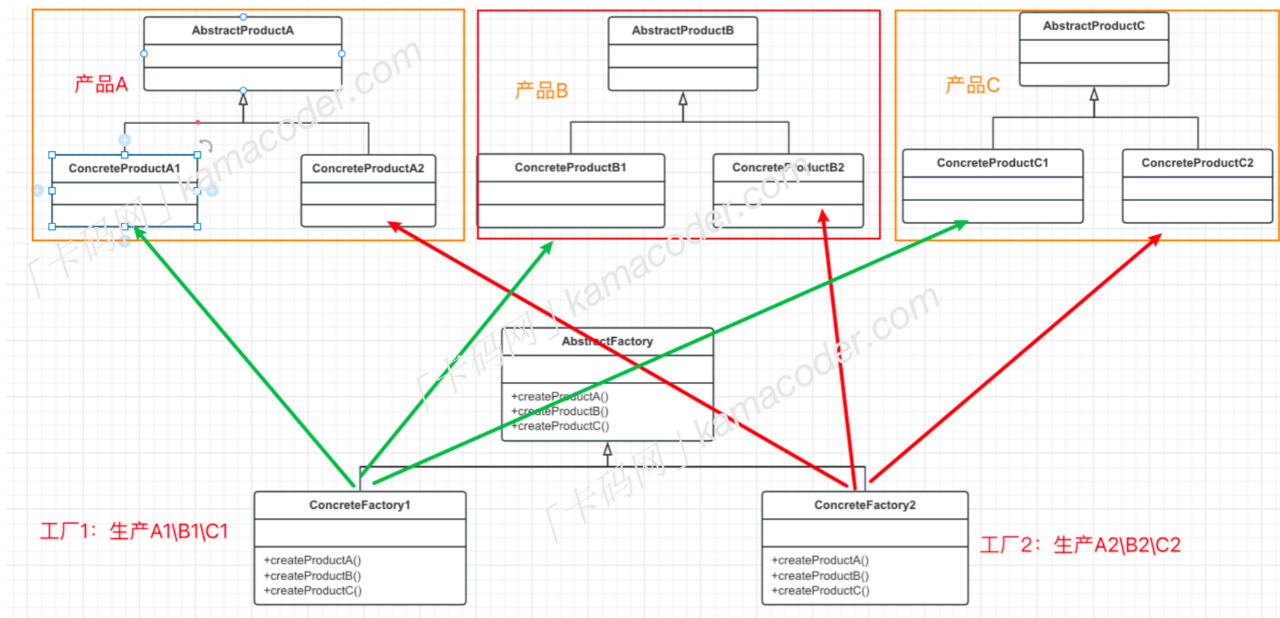
这就涉及到创建“多类”对象了，在工厂方法模式中，每个具体工厂只负责创建**单一**的产品。但是如果有多类产品呢，比如说“手机”，一个品牌的手机有高端机、中低端机之分，这些具体的产品都需要建立一个单独的工厂类，但是它们都是相互关联的，都共同属于同一个品牌，这就可以使用到【抽象工厂模式】。

抽象工厂模式可以确保一系列相关的产品被一起创建，这些产品能够相互配合使用，再举个例子，有一些家具，比如沙发、茶几、椅子，都具有古典风格的和现代风格的，抽象工厂模式可以将生产现代风格的家具放在一个工厂类中，将生产古典风格的家具放在另一个工厂类中，这样每个工厂类就可以生产一系列的家具。

基本结构

抽象工厂模式包含多个抽象产品接口，多个具体产品类，一个抽象工厂接口和多个具体工厂，每个具体工厂负责创建一组相关的产品。

- 抽象产品接口`AbstractProduct`：定义产品的接口，可以定义多个抽象产品接口，比如说沙发、椅子、茶几都是抽象产品。
- 具体产品类`ConcreteProduct`：实现抽象产品接口，产品的具体实现，古典风格和沙发和现代风格的沙发都是具体产品。
- 抽象工厂接口`AbstractFactory`：声明一组用于创建产品的方法，每个方法对应一个产品。
- 具体工厂类`ConcreteFactory`： 实现抽象工厂接口，负责创建一组具体产品的对象，在本例中，生产古典风格的工厂和生产现代风格的工厂都是具体实例。



在上面的图示中： `AbstractProductA/B/C` 就是抽象产品， `ConcreteProductA2/A2/B1/B2/C1/C2`就是抽象产品的实现， `AbstractFactory`定义了抽象工厂接口，接口里的方法用于创建具体的产品，而`ConcreteFactory`就是具体工厂类，可以创建一组相关的产品。

基本实现

想要实现抽象工厂模式，需要遵循以下步骤：

- 定义抽象产品接口（可以有多个），接口中声明产品的公共方法。
- 实现具体产品类，在类中实现抽象产品接口中的方法。
- 定义抽象工厂接口，声明一组用于创建产品的方法。
- 实现具体工厂类，分别实现抽象工厂接口中的方法，每个方法负责创建一组相关的产品。
- 在客户端中使用抽象工厂和抽象产品，而不直接使用具体产品的类名。

```java
// 1. 定义抽象产品
// 抽象产品A
interface ProductA {
    void display();
}

// 抽象产品B
interface ProductB {
    void show();
}

// 2. 实现具体产品类
// 具体产品A1
class ConcreteProductA1 implements ProductA {
    @Override
    public void display() {
        System.out.println("Concrete Product A1");
    }
}

// 具体产品A2
class ConcreteProductA2 implements ProductA {
    @Override
    public void display() {
        System.out.println("Concrete Product A2");
    }
}

// 具体产品B1
class ConcreteProductB1 implements ProductB {
    @Override
    public void show() {
        System.out.println("Concrete Product B1");
    }
}

// 具体产品B2
class ConcreteProductB2 implements ProductB {
    @Override
    public void show() {
        System.out.println("Concrete Product B2");
    }
}

// 3. 定义抽象工厂接口
interface AbstractFactory {
    ProductA createProductA();
    ProductB createProductB();
}
```

```java
// 4. 实现具体工厂类
// 具体工厂1，生产产品A1和B1
class ConcreteFactory1 implements AbstractFactory {
    @Override
    public ProductA createProductA() {
        return new ConcreteProductA1();
    }

    @Override
    public ProductB createProductB() {
        return new ConcreteProductB1();
    }
}

// 具体工厂2,生产产品A2和B2
class ConcreteFactory2 implements AbstractFactory {
    @Override
    public ProductA createProductA() {
        return new ConcreteProductA2();
    }

    @Override
    public ProductB createProductB() {
        return new ConcreteProductB2();
    }
}

// 客户端代码
public class AbstractFactoryExample {
    public static void main(String[] args) {
        // 使用工厂1创建产品A1和产品B1
        AbstractFactory factory1 = new ConcreteFactory1();
        ProductA productA1 = factory1.createProductA();
        ProductB productB1 = factory1.createProductB();
        productA1.display();
        productB1.show();

        // 使用工厂2创建产品A2和产品B2
        AbstractFactory factory2 = new ConcreteFactory2();
        ProductA productA2 = factory2.createProductA();
        ProductB productB2 = factory2.createProductB();
        productA2.display();
        productB2.show();
    }
}
```

应用场景

抽象工厂模式能够保证一系列相关的产品一起使用，并且在不修改客户端代码的情况下，可以方便地替换整个产品系列。但是当需要增加新的产品类时，除了要增加新的具体产品类，还需要修改抽象工厂接口及其所有的具体工厂类，扩展性相对较差。因此抽象工厂模式特别适用于一系列相关或相互依赖的产品被一起创建的情况，典型的应用场景是**使用抽象工厂模式来创建与不同数据库的连接对象。**

简单工厂、工厂方法、抽象工厂的区别

- 简单工厂模式：一个工厂方法创建所有具体产品
- 工厂方法模式：一个工厂方法创建一个具体产品
- 抽象工厂模式：一个工厂方法可以创建一类具体产品

本题代码

```java
import java.util.Scanner;

// 抽象椅子接口
interface Chair {
    void showInfo();
}

// 具体现代风格椅子
class ModernChair implements Chair {
    @Override
    public void showInfo() {
        System.out.println("modern chair");
    }
}

// 具体古典风格椅子
class ClassicalChair implements Chair {
    @Override
    public void showInfo() {
        System.out.println("classical chair");
    }
}

// 抽象沙发接口
interface Sofa {
    void displayInfo();
}

// 具体现代风格沙发
class ModernSofa implements Sofa {
    @Override
    public void displayInfo() {
        System.out.println("modern sofa");
    }
```

```java
}

// 具体古典风格沙发
class ClassicalSofa implements Sofa {
    @Override
    public void displayInfo() {
        System.out.println("classical sofa");
    }
}

// 抽象家居工厂接口
interface FurnitureFactory {
    Chair createChair();
    Sofa createSofa();
}

// 具体现代风格家居工厂
class ModernFurnitureFactory implements FurnitureFactory {
    @Override
    public Chair createChair() {
        return new ModernChair();
    }

    @Override
    public Sofa createSofa() {
        return new ModernSofa();
    }
}

// 具体古典风格家居工厂
class ClassicalFurnitureFactory implements FurnitureFactory {
    @Override
    public Chair createChair() {
        return new ClassicalChair();
    }

    @Override
    public Sofa createSofa() {
        return new ClassicalSofa();
    }
}

public class FurnitureOrderSystem {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取订单数量
        int N = scanner.nextInt();
```

```java
        // 处理每个订单
        for (int i = 0; i < N; i++) {
            // 读取家具类型
            String furnitureType = scanner.next();

            // 创建相应风格的家居装饰品工厂
            FurnitureFactory factory;
            if (furnitureType.equals("modern")) {
                factory = new ModernFurnitureFactory();
            } else if (furnitureType.equals("classical")) {
                factory = new ClassicalFurnitureFactory();
            }

            // 根据工厂生产椅子和沙发
            Chair chair = factory.createChair();
            Sofa sofa = factory.createSofa();

            // 输出家具信息
            chair.showInfo();
            sofa.displayInfo();
        }
    }
}
```

建造者模式

题目链接

[建造者模式-自行车加工]

什么是建造者模式

建造者模式（也被成为生成器模式），是一种创建型设计模式，软件开发过程中有的时候需要创建很复杂的对象，而建造者模式的主要思想是**将对象的构建过程分为多个步骤，并为每个步骤定义一个抽象的接口。具体的构建过程由实现了这些接口的具体建造者类来完成。**同时有一个指导者类负责协调建造者的工作，按照一定的顺序或逻辑来执行构建步骤，最终生成产品。

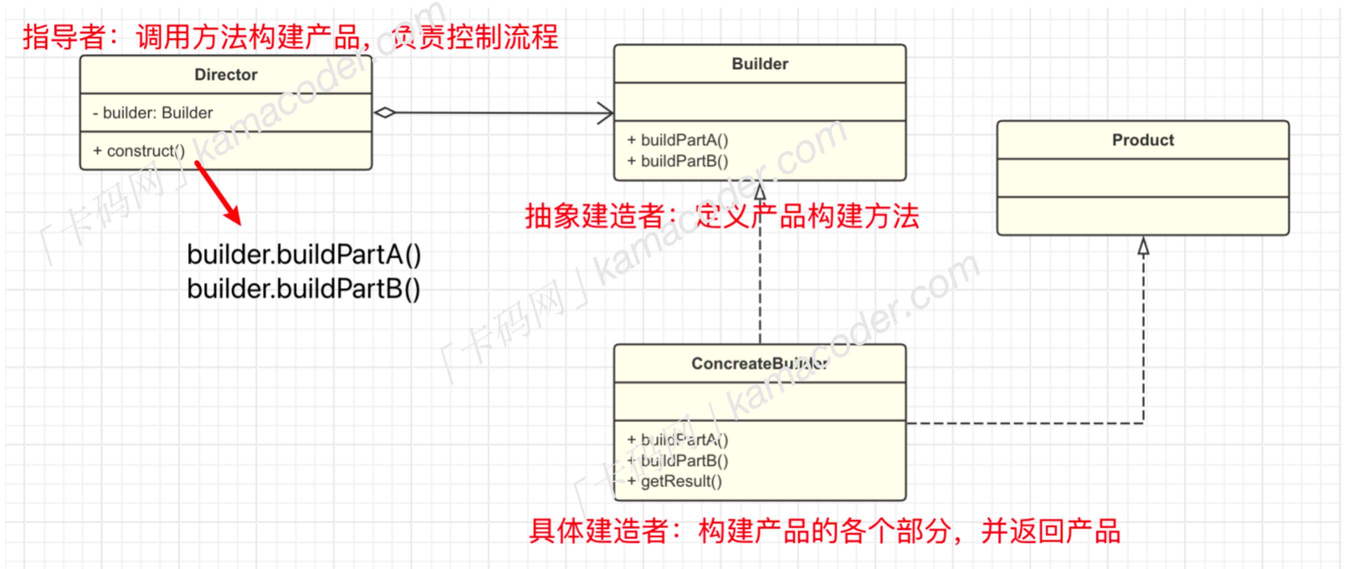
举个例子，假如我们要创建一个计算机对象，计算机由很多组件组成，例如 CPU、内存、硬盘、显卡等。每个组件可能有不同的型号、配置和制造，这个时候计算机就可以被视为一个复杂对象，构建过程相对复杂，而我们使用建造者模式将计算机的构建过程封装在一个具体的建造者类中，而指导者类则负责指导构建的步骤和顺序。每个具体的建造者类可以负责构建不同型号或配置的计算机，客户端代码可以通过选择不同的建造者来创建不同类型的计算机，这样就可以根据需要构建不同表示的复杂对象，更加灵活。

基本结构

建造者模式有下面几个关键角色：

- 产品`Product`： 被构建的复杂对象, 包含多个组成部分。
- 抽象建造者`Builder`: 定义构建产品各个部分的抽象接口和一个返回复杂产品的方法`getResult`
- 具体建造者`Concrete Builder`： 实现抽象建造者接口，构建产品的各个组成部分，并提供一个方法返回最终的产品。
- 指导者`Director`： 调用具体建造者的方法，按照一定的顺序或逻辑来构建产品。

在客户端中，通过指导者来构建产品，而并不和具体建造者进行直接的交互。



简易实现

建造者模式的实现步骤通常包括以下几个阶段

1. 定义产品类：产品类应该包含多个组成部分，这些部分的属性和方法构成了产品的接口

```
// 产品类
class Product {
    private String part1;
    private String part2;

    public void setPart1(String part1) {
        this.part1 = part1;
    }

    public void setPart2(String part2) {
        this.part2 = part2;
    }

    // 其他属性和方法
}
```

2. 定义抽象建造者接口：创建一个接口，包含构建产品各个部分的抽象方法。这些方法通常用于设置产品的各个属性。

```
// 抽象建造者接口
interface Builder {
    void buildPart1(String part1);
    void buildPart2(String part2);
    Product getResult();
}
```

3. 创建具体建造者：实现抽象建造者接口，构建具体的产品。

```
// 具体建造者类
class ConcreteBuilder implements Builder {
    private Product product = new Product();

    @Override
    public void buildPart1(String part1) {
        product.setPart1(part1);
    }

    @Override
    public void buildPart2(String part2) {
        product.setPart2(part2);
    }

    @Override
    public Product getResult() {
        return product;
    }
}
```

4. 定义`Director`类：指导者类来控制构建产品的顺序和步骤。

```
// 指导者类
class Director {
    private Builder builder;

    public Director(Builder builder) {
        this.builder = builder;
    }
  // 调用方法构建产品
    public void construct() {
        builder.buildPart1("Part1");
        builder.buildPart2("Part2");
    }
}
```

5. 客户端使用建造者模式：在客户端中创建【具体建造者对象】和【指导者对象】，通过指导者来构建产品。

```
// 客户端代码
public class Main{
    public static void main(String[] args) {
        // 创建具体建造者
        Builder builder = new ConcreteBuilder();

        // 创建指导者
        Director director = new Director(builder);

        // 指导者构建产品
        director.construct();

        // 获取构建好的产品
        Product product = builder.getResult();

        // 输出产品信息
        System.out.println(product);
    }
}
```

使用场景

使用建造者模式有下面几处优点：

- 使用建造者模式可以**将一个复杂对象的构建与其表示分离，**通过将构建复杂对象的过程抽象出来，可以使客户端代码与具体的构建过程解耦
- **同样的构建过程可以创建不同的表示**，可以有多个具体的建造者(相互独立），可以更加灵活地创建不同组合的对象。

对应的，建造者模式适用于复杂对象的创建，当对象构建过程相对复杂时可以考虑使用建造者模式，但是当产品的构建过程发生变化时，可能需要同时修改指导类和建造者类，这就使得重构变得相对困难。

建造者模式在现有的工具和库中也有着广泛的应用，比如JUnit 中的测试构建器`TestBuilder`就采用了建造者模式，用于构建测试对象。

本题代码

```
import java.util.Scanner;

// 自行车产品
class Bike {
    private String frame;
    private String tires;

    public void setFrame(String frame) {
        this.frame = frame;
    }
```

```java
    public void setTires(String tires) {
        this.tires = tires;
    }

    @Override
    public String toString() {
        return frame + " " + tires;
    }
}

// 自行车建造者接口
interface BikeBuilder {
    void buildFrame();
    void buildTires();
    Bike getResult();
}

// 山地自行车建造者
class MountainBikeBuilder implements BikeBuilder {
    private Bike bike;

    public MountainBikeBuilder() {
        this.bike = new Bike();
    }

    @Override
    public void buildFrame() {
        bike.setFrame("Aluminum Frame");
    }

    @Override
    public void buildTires() {
        bike.setTires("Knobby Tires");
    }

    @Override
    public Bike getResult() {
        return bike;
    }
}

// 公路自行车建造者
class RoadBikeBuilder implements BikeBuilder {
    private Bike bike;

    public RoadBikeBuilder() {
        this.bike = new Bike();
    }
```

```java
    @Override
    public void buildFrame() {
        bike.setFrame("Carbon Frame");
    }

    @Override
    public void buildTires() {
        bike.setTires("Slim Tires");
    }

    @Override
    public Bike getResult() {
        return bike;
    }
}

// 自行车Director，负责构建自行车
class BikeDirector {
    public Bike construct(BikeBuilder builder) {
        builder.buildFrame();
        builder.buildTires();
        return builder.getResult();
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int N = scanner.nextInt();  // 订单数量
        scanner.nextLine();

        BikeDirector director = new BikeDirector();

        for (int i = 0; i < N; i++) {
            String bikeType = scanner.nextLine();

            BikeBuilder builder;
            // 根据输入类别，创建不同类型的具体建造者
            if (bikeType.equals("mountain")) {
                builder = new MountainBikeBuilder();
            } else {
                builder = new RoadBikeBuilder();
            }
      // Director负责指导生产产品
            Bike bike = director.construct(builder);
            System.out.println(bike);
        }
```

```
    }
}
```

原型模式

题目链接

[原型模式-矩形原型]

什么是原型模式

原型模式一种创建型设计模式，该模式的核心思想是基于现有的对象创建新的对象，而不是从头开始创建。

在原型模式中，通常有一个原型对象，它被用作创建新对象的模板。新对象通过复制原型对象的属性和状态来创建，而无需知道具体的创建细节。

为什么要使用原型模式

如果一个对象的创建过程比较复杂时（比如需要经过一系列的计算和资源消耗），那每次创建该对象都需要消耗资源，而通过原型模式就可以复制现有的一个对象来迅速创建/克隆一个新对象，不必关心具体的创建细节，可以降低对象创建的成本。

下面是一个简短的Python代码示例了模拟了上面的问题：

```python
import copy

class ComplexObject:
    def __init__(self, data):
        # 耗时的资源型操作
        self.data = data

    def clone(self):
      # 复制
        return copy.deepcopy(self)
# 创建原型对象
original_object = ComplexObject(data="large date")
# 创建新对象，直接拷贝原对象
new_object = original_object.clone()
```

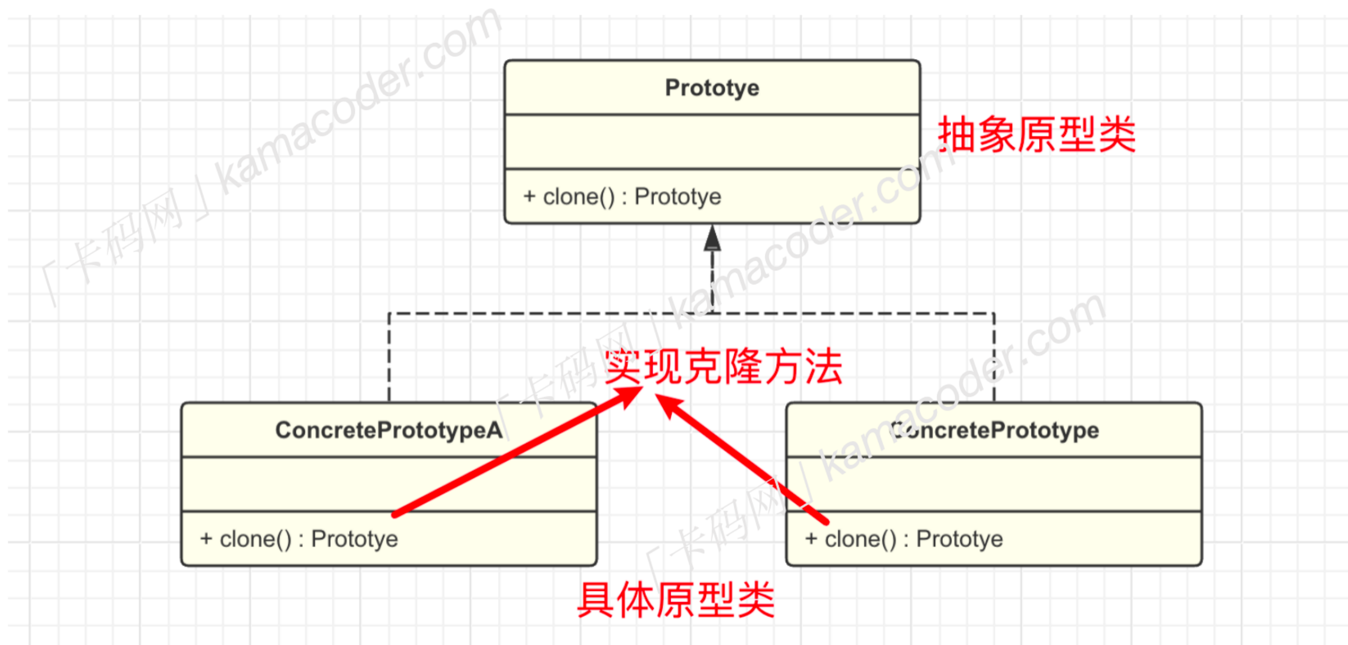
原型模式的基本结构

实现原型模式需要给【原型对象】声明一个克隆方法，执行该方法会创建一个当前类的新对象，并将原始对象中的成员变量复制到新生成的对象中，而不必实例化。并且在这个过程中只需要调用原型对象的克隆方法，而无需知道原型对象的具体类型。

原型模式包含两个重点模块：

- 抽象原型接口`prototype`: 声明一个克隆自身的方法`clone`
- 具体原型类`ConcretePrototype`：实现`clone`方法，复制当前对象并返回一个新对象。

在客户端代码中，可以声明一个具体原型类的对象，然后调用`clone()`方法复制原对象生成一个新的对象。



原型模式的基本实现

原型模式的实现过程即上面描述模块的实现过程：

- 创建一个抽象类或接口，声明一个克隆方法`clone`
- 实现具体原型类，重写克隆方法
- 客户端中实例化具体原型类的对象，并调用其克隆方法来创建新的对象。

```java
// 1. 定义抽象原型类
public abstract class Prototype implements Cloneable {
    public abstract Prototype clone();
}

// 2. 创建具体原型类
public class ConcretePrototype extends Prototype {
    private String data;

    public ConcretePrototype(String data) {
        this.data = data;
    }

    @Override
    public Prototype clone() {
        return new ConcretePrototype(this.data);
    }

    public String getData() {
        return data;
    }
}
```

```java
// 3. 客户端代码
public class Client {
    public static void main(String[] args) {
        // 创建原型对象
        Prototype original = new ConcretePrototype("Original Data");

        // 克隆原型对象
        Prototype clone = original.clone();

        // 输出克隆对象的数据
        System.out.println("Clone Data: " + ((ConcretePrototype) clone).getData());
    }
}
```

什么时候实现原型模式

相比于直接实例化对象，通过原型模式复制对象可以减少资源消耗，提高性能，尤其在对象的创建过程复杂或对象的创建代价较大的情况下。当需要频繁创建相似对象、并且可以通过克隆避免重复初始化工作的场景时可以考虑使用原型模式，在克隆对象的时候还可以动态地添加或删除原型对象的属性，创造出相似但不完全相同的对象，提高了灵活性。

但是使用原型模式也需要考虑到如果对象的内部状态包含了引用类型的成员变量，那么实现深拷贝就会变得较为复杂，需要考虑引用类型对象的克隆问题。

原型模式在现有的很多语言中都有应用，比如以下几个经典例子。

- Java 提供了 `Object` 类的 `clone()` 方法，可以实现对象的浅拷贝。类需要实现 `Cloneable` 接口并重写 `clone()` 方法。
- 在 .NET 中，`ICloneable` 接口提供了 `Clone` 方法，可以用于实现对象的克隆。
- Spring 框架中的 Bean 的作用域之一是原型作用域（Prototype Scope），在这个作用域下，Spring 框架会为每次请求创建一个新的 Bean 实例，类似于原型模式。

本题代码

```java
import java.util.Scanner;

// 抽象原型类
abstract class Prototype implements Cloneable {
    public abstract Prototype clone();
    public abstract String getDetails();

    // 公共的 clone 方法
    public Prototype clonePrototype() {
        try {
            return (Prototype) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
```

```java
        }
    }
}

// 具体矩形原型类
class RectanglePrototype extends Prototype {
    private String color;
    private int width;
    private int height;

    // 构造方法
    public RectanglePrototype(String color, int width, int height) {
        this.color = color;
        this.width = width;
        this.height = height;
    }

    // 克隆方法
    @Override
    public Prototype clone() {
        return clonePrototype();
    }

    // 获取矩形的详细信息
    @Override
    public String getDetails() {
        return "Color: " + color + ", Width: " + width + ", Height: " + height;
    }
}

// 客户端程序
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取需要创建的矩形数量
        int N = scanner.nextInt();

        // 读取每个矩形的属性信息并创建矩形对象
        for (int i = 0; i < N; i++) {
            String color = scanner.next();
            int width = scanner.nextInt();
            int height = scanner.nextInt();

            // 创建原型对象
            Prototype originalRectangle = new RectanglePrototype(color, width, height);

            // 克隆对象并输出详细信息
            Prototype clonedRectangle = originalRectangle.clone();
```

```
            System.out.println(clonedRectangle.getDetails());
        }
    }
}
```

适配器模式

题目链接

[适配器模式-扩展坞]

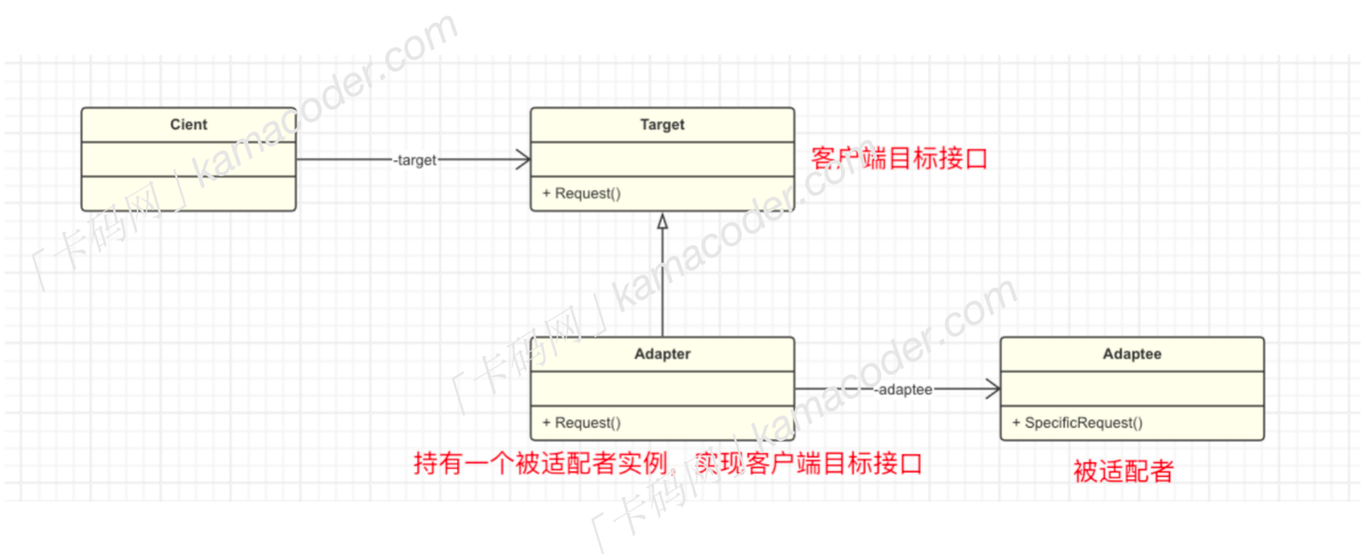
什么是适配器

适配器模式`Adapter`是一种结构型设计模式，它可以将一个类的接口转换成客户希望的另一个接口，主要目的是充当两个不同接口之间的桥梁，使得原本接口不兼容的类能够一起工作。

基本结构

适配器模式分为以下几个基本角色：

> 可以把适配器模式理解成拓展坞，起到转接的作用，原有的接口是USB，但是客户端需要使用`type-c`，便使用拓展坞提供一个`type-c`接口给客户端使用



- 目标接口`Target`：客户端希望使用的接口
- 适配器类`Adapter`：实现客户端使用的目标接口，持有一个需要适配的类实例。
- 被适配者`Adaptee`：需要被适配的类

这样，客户端就可以使用目标接口，而不需要对原来的`Adaptee`进行修改，`Adapter`起到一个转接扩展的作用。

基本实现

```
// 目标接口
interface Target {
    void request();
}

// 被适配者类
class Adaptee {
    void specificRequest() {
        System.out.println("Specific request");
    }
}

// 适配器类
class Adapter implements Target {
    // 持有一个被适配者实例
    private Adaptee adaptee;

    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    @Override
    public void request() {
        // 调用被适配者类的方法
        adaptee.specificRequest();
    }
}

// 客户端代码
public class Client {
    public static void main(String[] args) {
        Target target = new Adapter(new Adaptee());
        target.request();
    }
}
```

应用场景

在开发过程中，适配器模式往往扮演者“补救”和“扩展”的角色：

- 当使用一个已经存在的类，但是它的接口与你的代码不兼容时，可以使用适配器模式。
- 在系统扩展阶段需要增加新的类时，并且类的接口和系统现有的类不一致时，可以使用适配器模式。

使用适配器模式可以将客户端代码与具体的类解耦，客户端不需要知道被适配者的细节，客户端代码也不需要修改，这使得它具有良好的扩展性，但是这也势必导致系统变得更加复杂。

具体来说，适配器模式有着以下应用：

- 不同的项目和库可能使用不同的日志框架，不同的日志框架提供的API也不同，因此引入了适配器模式使得不同的API适配为统一接口。
- Spring MVC中，`HandlerAdapter` 接口是适配器模式的一种应用。它负责将处理器（Handler）适配到框架中，使得不同类型的处理器能够统一处理请求。
- 在 `.NET` 中，`DataAdapter` 用于在数据源（如数据库）和 `DataSet` 之间建立适配器，将数据从数据源适配到 `DataSet` 中，以便在.NET应用程序中使用。

本题代码

```
// 测试程序
import java.util.Scanner;
// USB 接口
interface USB {
    void charge();
}

// TypeC 接口
interface TypeC {
    void chargeWithTypeC();
}

// 适配器类
class TypeCAdapter implements USB {
    private TypeC typeC;

    public TypeCAdapter(TypeC typeC) {
        this.typeC = typeC;
    }

    @Override
    public void charge() {
        typeC.chargeWithTypeC();
    }
}

// 新电脑类，使用 TypeC 接口
class NewComputer implements TypeC {
    @Override
    public void chargeWithTypeC() {
        System.out.println("TypeC");
    }
}

// 适配器充电器类，使用 USB 接口
class AdapterCharger implements USB {
    @Override
    public void charge() {
        System.out.println("USB Adapter");
```

```java
    }
}



public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取连接次数
        int N = scanner.nextInt();
        scanner.nextLine(); // 消耗换行符

        for (int i = 0; i < N; i++) {
            // 读取用户选择
            int choice = scanner.nextInt();

            // 根据用户的选择创建相应对象
            if (choice == 1) {
                TypeC newComputer = new NewComputer();
                newComputer.chargeWithTypeC();
            } else if (choice == 2) {
                USB usbAdapter = new AdapterCharger();
                usbAdapter.charge();
            }
        }
        scanner.close();
    }
}
```

代理模式

题目链接

[代理模式-小明买房子]

基本概念

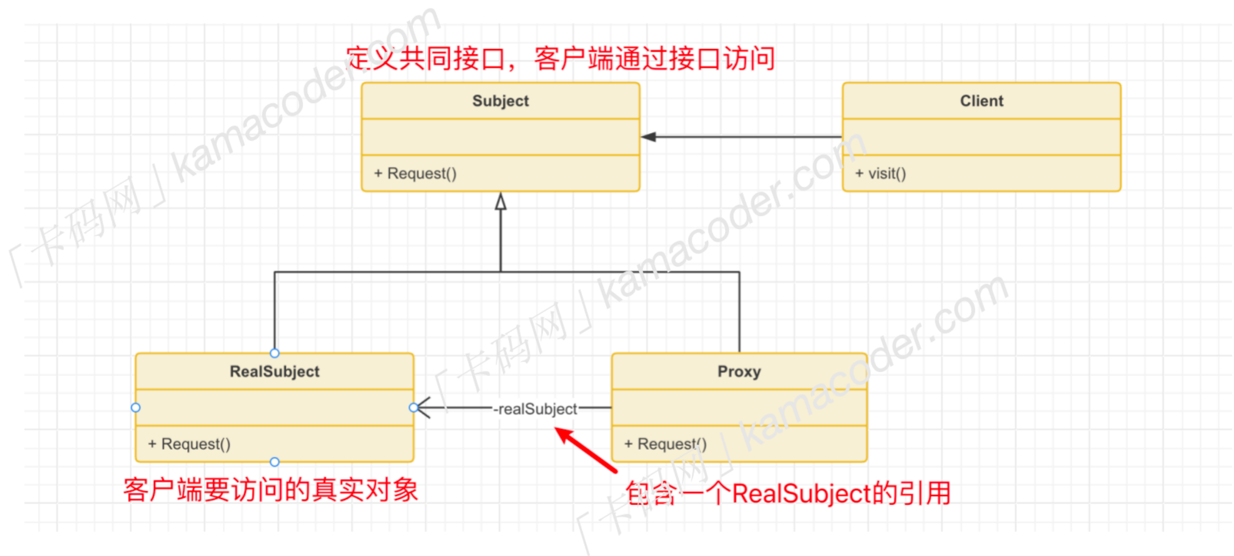
代理模式`Proxy Pattern`是一种结构型设计模式，用于控制对其他对象的访问。

在代理模式中，允许一个对象（代理）充当另一个对象（真实对象）的接口，以控制对这个对象的访问。通常用于在访问某个对象时引入一些间接层(中介的作用)，这样可以在访问对象时添加额外的控制逻辑，比如限制访问权限，延迟加载。

比如说有一个文件加载的场景，为了避免直接访问“文件”对象，我们可以新增一个代理对象，代理对象中有一个对“文件对象”的引用，在代理对象的 `load` 方法中，可以在访问真实的文件对象之前进行一些操作，比如权限检查，然后调用真实文件对象的 `load` 方法，最后在访问真实对象后进行其他操作，比如记录访问日志。

基本结构

代理模式的主要角色有：



- Subject（抽象主题）： 抽象类，通过接口或抽象类声明真实主题和代理对象实现的业务方法。
- RealSubject（真实主题）：定义了Proxy所代表的真实对象，是客户端最终要访问的对象。
- Proxy（代理）：包含一个引用，该引用可以是RealSubject的实例，控制对RealSubject的访问，并可能负责创建和删除RealSubject的实例。

实现方式

代理模式的基本实现分为以下几个步骤：

1. 定义抽象主题，一般是接口或者抽象类，声明真实主题和代理对象实现的业务方法。

```
// 1. 定义抽象主题
interface Subject {
    void request();
}
```

2. 定义真实主题，实现抽象主题中的具体业务

```
// 2. 定义真实主题
class RealSubject implements Subject {
    @Override
    public void request() {
        System.out.println("RealSubject handles the request.");
    }
}
```

3. 定义代理类，包含对`RealSubject`的引用，并提供和真实主题相同的接口，这样代理就可以替代真实主题，并对真实主题进行功能扩展。

```java
// 3. 定义代理
class Proxy implements Subject {
    // 包含一个引用
    private RealSubject realSubject;

    @Override
    public void request() {
        // 在访问真实主题之前可以添加额外的逻辑
        if (realSubject == null) {
            realSubject = new RealSubject();
        }
        // 调用真实主题的方法
        realSubject.request();

        // 在访问真实主题之后可以添加额外的逻辑
    }
}
```

4. 客户端使用代理

```java
// 4. 客户端使用代理
public class Main {
    public static void main(String[] args) {
        // 使用代理
        Subject proxy = new Proxy();
        proxy.request();
    }
}
```

使用场景

代理模式可以控制客户端对真实对象的访问，从而限制某些客户端的访问权限，此外代理模式还常用在访问真实对象之前或之后执行一些额外的操作（比如记录日志），对功能进行扩展。

以上特性决定了代理模式在以下几个场景中有着广泛的应用：

- 虚拟代理：当一个对象的创建和初始化比较昂贵时，可以使用虚拟代理，虚拟代理可以延迟对象的实际创建和初始化，只有在需要时才真正创建并初始化对象。
- 安全代理：安全代理可以根据访问者的权限决定是否允许访问真实对象的方法。

但是代理模式涉及到多个对象之间的交互，引入代理模式会增加系统的复杂性，在需要频繁访问真实对象时，还可能会有一些性能问题。

代理模式在许多工具和库中也有应用：

- Spring 框架的 AOP 模块使用了代理模式来实现切面编程。通过代理，Spring 能够在目标对象的方法执行前、执行后或抛出异常时插入切面逻辑，而不需要修改原始代码。
- Java 提供了动态代理机制，允许在运行时生成代理类。
- Android中的Glide框架使用了代理模式来实现图片的延迟加载。

本题代码

```java
import java.util.Scanner;

// 抽象主题
interface HomePurchase {
    void requestPurchase(int area);
}

//真实主题
class HomeBuyer implements HomePurchase {
    @Override
    public void requestPurchase(int area) {
        System.out.println("YES");
    }
}

// 代理类
class HomeAgentProxy implements HomePurchase {
    private HomeBuyer homeBuyer = new HomeBuyer();

    @Override
    public void requestPurchase(int area) {
        if (area > 100) {
            homeBuyer.requestPurchase(area);
        } else {
            System.out.println("NO");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        HomePurchase buyerProxy = new HomeAgentProxy();

        int n = scanner.nextInt();
        for (int i = 0; i < n; i++) {
            int area = scanner.nextInt();
            buyerProxy.requestPurchase(area);
        }

        scanner.close();
    }
}
```

扩展：代理模式和适配器模式有什么区别

代理模式的主要目的是控制对对象的访问。通常用于在访问真实对象时引入一些额外的控制逻辑，如权限控制、延迟加载等。

适配器模式的主要目的是使接口不兼容的对象能够协同工作。适配器模式允许将一个类的接口转换成另一个类的接口，使得不同接口的类可以协同工作。

装饰模式

题目链接

[装饰器模式-咖啡加糖]()

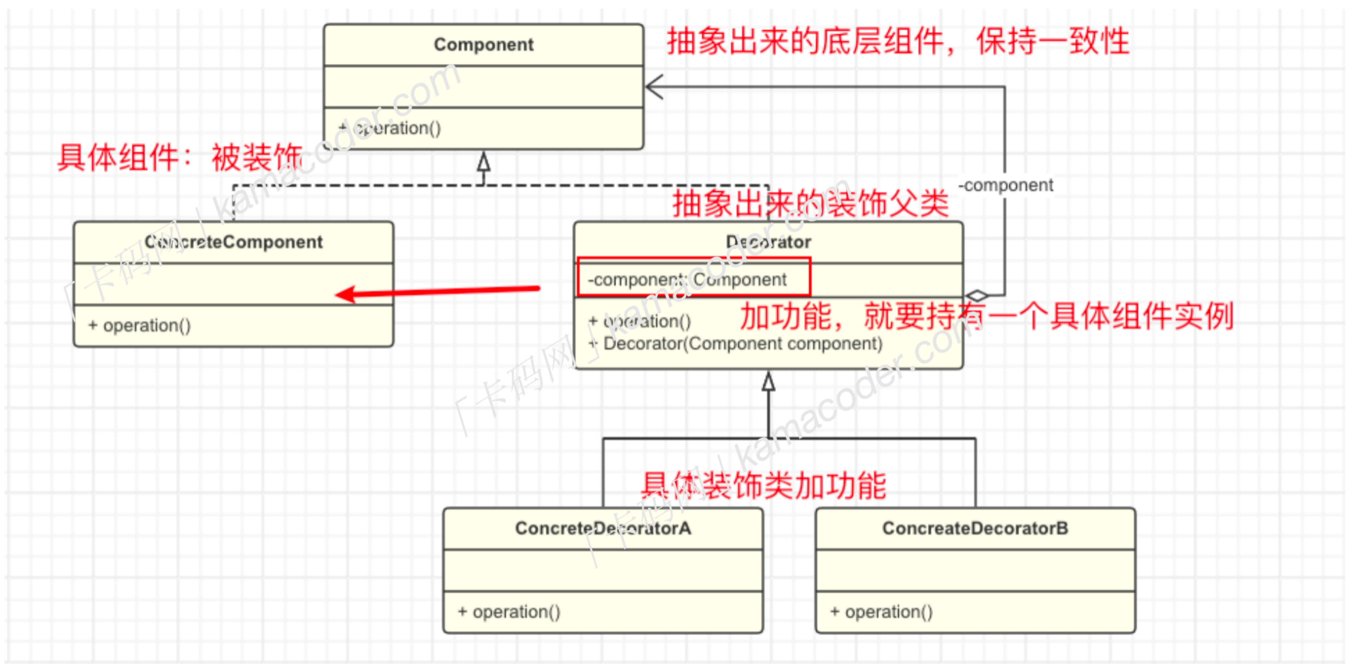
基本概念

通常情况下，扩展类的功能可以通过继承实现，但是扩展越多，子类越多，装饰模式（`Decorator Pattern`，结构型设计模式）可以在**不定义子类的情况下动态的给对象添加一些额外的功能。**具体的做法是将原始对象放入包含行为的特殊封装类(装饰类)，从而为原始对象动态添加新的行为，而无需修改其代码。

举个简单的例子，假设你有一个基础的图形类，你想要为图形类添加颜色、边框、阴影等功能，如果每个功能都实现一个子类，就会导致产生大量的类，这时就可以考虑使用装饰模式来动态地添加，而不需要修改图形类本身的代码，这样可以使得代码更加灵活、更容易维护和扩展。

基本结构：

装饰模式包含以下四个主要角色：



- 组件`Component`：通常是抽象类或者接口，是具体组件和装饰者的父类，定义了具体组件需要实现的方法，比如说我们定义`Coffee`为组件。
- 具体组件`ConcreteComponent`：实现了Component接口的具体类，是**被装饰的对象**。
- 装饰类`Decorator`：一个抽象类，给具体组件添加功能，但是具体的功能由其子类具体装饰者完成，持有一

个指向Component对象的引用。
- 具体装饰类`ConcreteDecorator`：扩展Decorator类，负责向Component对象添加新的行为，加牛奶的咖啡是一个具体装饰类，加糖的咖啡也是一个具体装饰类。

基本实现

装饰模式的实现包括以下步骤：

1. 定义Component接口

```
// 组件接口
public interface Component {
    void operation();
}
```

2. 实现 ConcreteComponent

```
// 具体组件
public class ConcreteComponent implements Component {
    @Override
    public void operation() {
        System.out.println("ConcreteComponent operation");
    }
}
```

3. 定义Decorator装饰类，继承自Component

```
// 定义一个抽象的装饰者类，继承自Component
public abstract class Decorator implements Component {
    protected Component component;

    public Decorator(Component component) {
        this.component = component;
    }

    @Override
    public void operation() {
        component.operation();
    }
}
```

4. 定义具体的装饰者实现，给具体组件对象添加功能。

```
// 具体的装饰者实现
public class ConcreteDecorator extends Decorator {
    public ConcreteDecorator(Component component) {
        super(component);
    }
```

```java
    // 根据需要添加额外的方法

    @Override
    public void operation() {
        // 可以在调用前后添加额外的行为
        System.out.println("Before operation in ConcreteDecorator");
        super.operation();
        System.out.println("After operation in ConcreteDecorator");
    }
}
```

5. 在客户端使用

```java
public class Main {
    public static void main(String[] args) {
        // 创建具体组件
        Component concreteComponent = new ConcreteComponent();

        // 使用具体装饰者包装具体组件
        Decorator decorator = new ConcreteDecorator(concreteComponent);

        // 调用操作
        decorator.operation();
    }
}
```

应用场景

装饰模式通常在以下几种情况使用：

- 当需要给一个现有类添加附加功能，但由于某些原因不能使用继承来生成子类进行扩充时，可以使用装饰模式。
- 动态的添加和覆盖功能：当对象的功能要求可以动态地添加，也可以再动态地撤销时可以使用装饰模式。

在Java的I/O库中，装饰者模式被广泛用于增强I/O流的功能。例如，`BufferedInputStream`和`BufferedOutputStream`这两个类提供了缓冲区的支持，通过在底层的输入流和输出流上添加缓冲区，提高了读写的效率，它们都是`InputStream`和`OutputStream`的装饰器。`BufferedReader`和`BufferedWriter`这两个类与`BufferedInputStream`和`BufferedOutputStream`类似，提供了字符流的缓冲功能，是Reader和Writer的装饰者。

本题代码

```java
import java.util.Scanner;

// 咖啡接口
interface Coffee {
    void brew();
```

```java
}

// 具体的黑咖啡类
class BlackCoffee implements Coffee {
    @Override
    public void brew() {
        System.out.println("Brewing Black Coffee");
    }
}

// 具体的拿铁类
class Latte implements Coffee {
    @Override
    public void brew() {
        System.out.println("Brewing Latte");
    }
}

// 装饰者抽象类
abstract class Decorator implements Coffee {
    protected Coffee coffee;

    public Decorator(Coffee coffee) {
        this.coffee = coffee;
    }

    @Override
    public void brew() {
        coffee.brew();
    }
}

// 具体的牛奶装饰者类
class MilkDecorator extends Decorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public void brew() {
        super.brew();
        System.out.println("Adding Milk");
    }
}

// 具体的糖装饰者类
class SugarDecorator extends Decorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
```

```java
    }

    @Override
    public void brew() {
        super.brew();
        System.out.println("Adding Sugar");
    }
}

// 客户端代码
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        while (scanner.hasNext()) {
            int coffeeType = scanner.nextInt();
            int condimentType = scanner.nextInt();

            // 根据输入制作咖啡
            Coffee coffee;
            if (coffeeType == 1) {
                coffee = new BlackCoffee();
            } else if (coffeeType == 2) {
                coffee = new Latte();
            } else {
                System.out.println("Invalid coffee type");
                continue;
            }

            // 根据输入添加调料
            if (condimentType == 1) {
                coffee = new MilkDecorator(coffee);
            } else if (condimentType == 2) {
                coffee = new SugarDecorator(coffee);
            } else {
                System.out.println("Invalid condiment type");
                continue;
            }

            // 输出制作过程
            coffee.brew();
        }
    }
}
```

外观模式

题目链接

[外观模式-电源开关]

基本概念

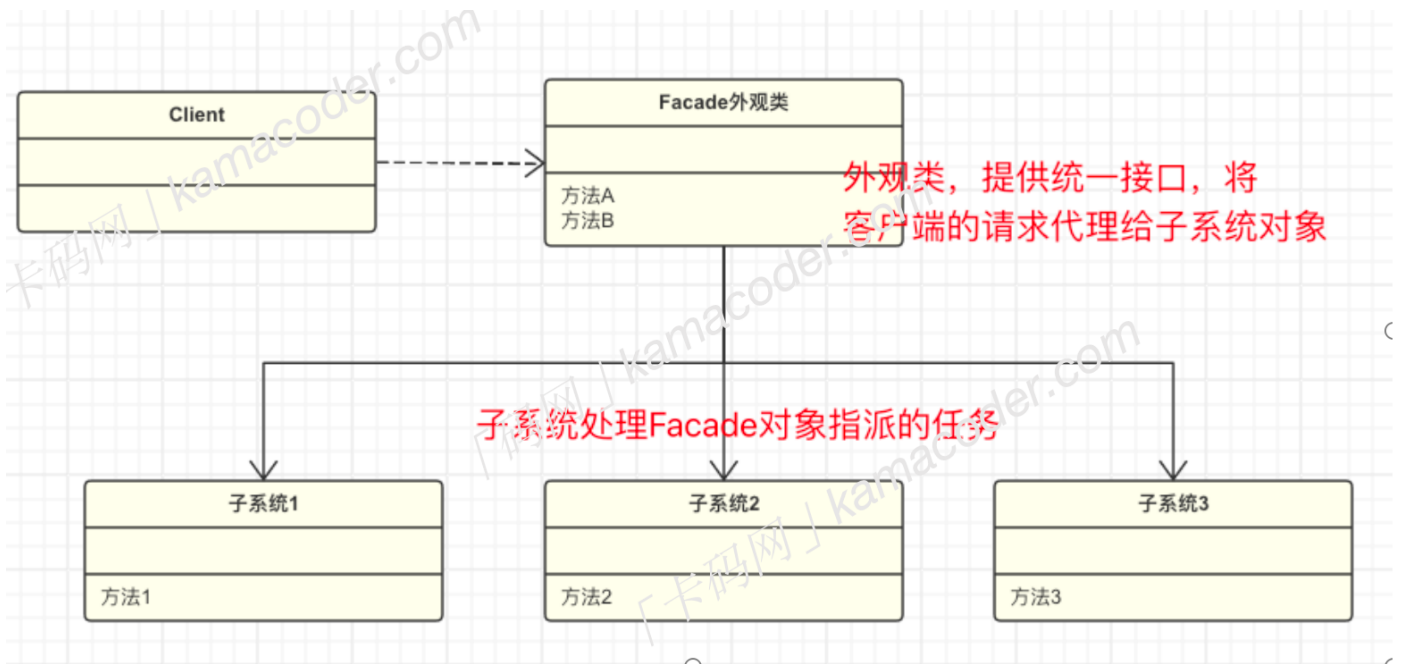
外观模式`Facade Pattern`，也被称为“门面模式”，是一种结构型设计模式，外观模式定义了一个高层接口，这个接口使得子系统更容易使用，同时也隐藏了子系统的复杂性。

> 门面模式可以将子系统关在“门里”隐藏起来，客户端只需要通过外观接口与外观对象进行交互，而不需要直接和多个子系统交互，无论子系统多么复杂，对于外部来说是隐藏的，这样可以降低系统的耦合度。

举个例子，假设你正在编写的一个模块用来处理文件读取、解析、存储，我们可以将这个过程拆成三部分，然后创建一个外观类，将文件系统操作、数据解析和存储操作封装在外观类中，为客户端提供一个简化的接口，如果后续需要修改文件处理的流程或替换底层子系统，也只需在外观类中进行调整，不会影响客户端代码。

基本结构

外观模式的基本结构比较简单，只包括“外观”和“子系统类”



- 外观类：对外提供一个统一的高层次接口，使复杂的子系统变得更易使用。
- 子系统类：实现子系统的功能，处理外观类指派的任务。

简易实现

下面使用Java代码实现外观模式的通用结构

```
// 子系统A
class SubsystemA {
    public void operationA() {
        System.out.println("SubsystemA operation");
    }
}
```

```java
// 子系统B
class SubsystemB {
    public void operationB() {
        System.out.println("SubsystemB operation");
    }
}

// 子系统C
class SubsystemC {
    public void operationC() {
        System.out.println("SubsystemC operation");
    }
}

// 外观类
class Facade {
    private SubsystemA subsystemA;
    private SubsystemB subsystemB;
    private SubsystemC subsystemC;

    public Facade() {
        this.subsystemA = new SubsystemA();
        this.subsystemB = new SubsystemB();
        this.subsystemC = new SubsystemC();
    }

    // 外观方法，封装了对子系统的操作
    public void facadeOperation() {
        subsystemA.operationA();
        subsystemB.operationB();
        subsystemC.operationC();
    }
}

// 客户端
public class Main {
    public static void main(String[] args) {
        // 创建外观对象
        Facade facade = new Facade();

        // 客户端通过外观类调用子系统的操作
        facade.facadeOperation();
    }
}
```

在上面的代码中， `Facade` 类是外观类，封装了对三个子系统 `SubSystem` 的操作。客户端通过调用外观类的方法来实现对子系统的访问，而不需要直接调用子系统的方法。

优缺点和使用场景

外观模式通过提供一个简化的接口，隐藏了系统的复杂性，降低了客户端和子系统之间的耦合度，客户端不需要了解系统的内部实现细节，也不需要直接和多个子系统交互，只需要通过外观接口与外观对象进行交互。

但是如果需要添加新的子系统或修改子系统的行为，就可能需要修改外观类，这违背了“开闭原则”。

外观模式的应用也十分普遍，下面几种情况都使用了外观模式来进行简化。

- Spring框架是一个广泛使用外观模式的例子。Spring框架提供了一个大量的功能，包括依赖注入、面向切面编程（AOP）、事务管理等。Spring的`ApplicationContext`可以看作是外观，隐藏了底层组件的复杂性，使得开发者可以更轻松地使用Spring的功能。
- JDBC提供了一个用于与数据库交互的接口。`DriverManager`类可以看作是外观，它简化了数据库驱动的加载和连接的过程，隐藏了底层数据库连接的复杂性。
- Android系统的API中也使用了外观模式。例如，`Activity`类提供了一个外观，使得开发者可以更容易地管理应用的生命周期，而无需关心底层的事件和状态管理。

本题代码

```java
import java.util.Scanner;

class AirConditioner {
    public void turnOff() {
        System.out.println("Air Conditioner is turned off.");
    }
}
class DeskLamp {
    public void turnOff() {
        System.out.println("Desk Lamp is turned off.");
    }
}
class Television {
    public void turnOff() {
        System.out.println("Television is turned off.");
    }
}

class PowerSwitchFacade {
    private DeskLamp deskLamp;
    private AirConditioner airConditioner;
    private Television television;

    public PowerSwitchFacade() {
        this.deskLamp = new DeskLamp();
        this.airConditioner = new AirConditioner();
        this.television = new Television();
    }

    public void turnOffDevice(int deviceCode) {
```

```
        switch (deviceCode) {
            case 1:
                airConditioner.turnOff();
                break;
            case 2:
                deskLamp.turnOff();
                break;
            case 3:
                television.turnOff();
                break;
            case 4:
                System.out.println("All devices are off.");
                break;
            default:
                System.out.println("Invalid device code.");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取输入
        int n = scanner.nextInt();
        int[] input = new int[n];

        for (int i = 0; i < n; i++) {
            input[i] = scanner.nextInt();
        }

        // 创建电源总开关外观
        PowerSwitchFacade powerSwitch = new PowerSwitchFacade();

        // 执行操作
        for (int i = 0; i < n; i++) {
            powerSwitch.turnOffDevice(input[i]);
        }
    }
}
```

桥接模式

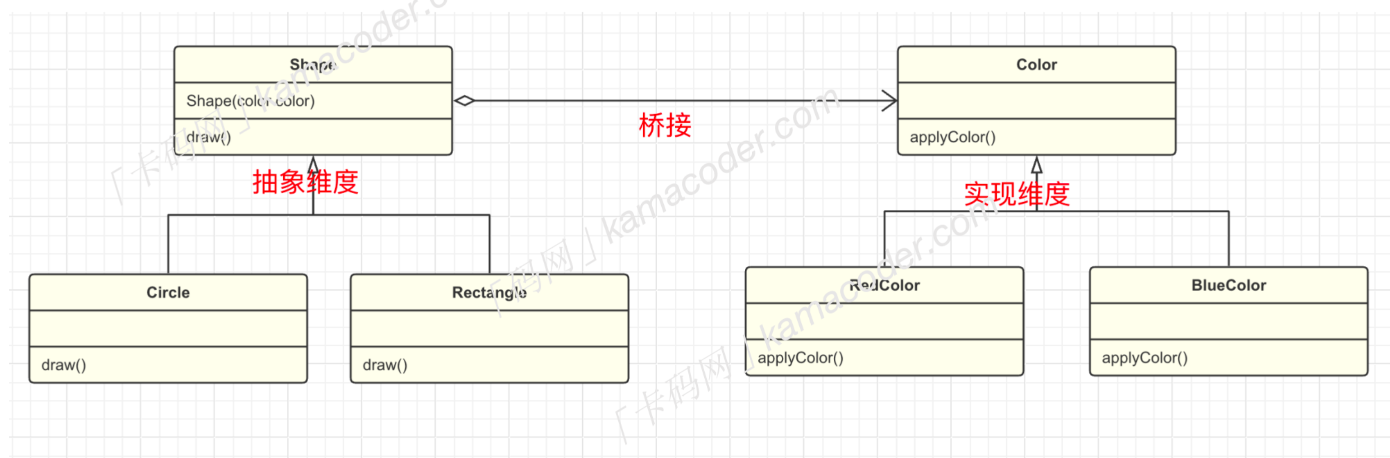
题目链接

[桥接模式-万能遥控器]

基本概念

桥接模式（Bridge Pattern）是一种结构型设计模式，它的UML图很像一座桥，它通过将【抽象部分】与【实现部分】分离，使它们可以独立变化，从而达到降低系统耦合度的目的。桥接模式的主要目的是通过组合建立两个类之间的联系，而不是继承的方式。

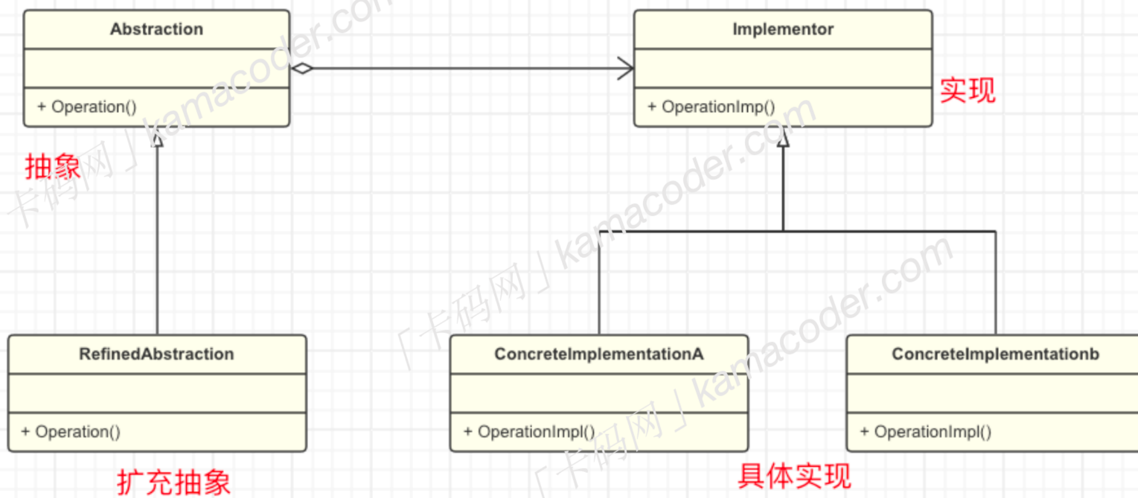
举个简单的例子，图形编辑器中，每一种图形都需要蓝色、红色、黄色不同的颜色，如果不使用桥接模式，可能需要为每一种图形类型和每一种颜色都创建一个具体的子类，而使用桥接模式可以将图形和颜色两个维度分离，两个维度都可以独立进行变化和扩展，如果要新增其他颜色，只需添加新的 `Color` 子类，不影响图形类；反之亦然。



基本结构

桥接模式的基本结构分为以下几个角色：

- 抽象 `Abstraction`：一般是抽象类，定义抽象部分的接口，维护一个对【实现】的引用。
- 修正抽象 `RefinedAbstaction`：对抽象接口进行扩展，通常对抽象化的不同维度进行变化或定制。
- 实现 `Implementor`： 定义实现部分的接口，提供具体的实现。这个接口通常是抽象化接口的实现。
- 具体实现 `ConcreteImplementor`：实现实现化接口的具体类。这些类负责实现实现化接口定义的具体操作。



再举个例子，遥控器就是抽象接口，它具有开关电视的功能，修正抽象就是遥控器的实例，对遥控器的功能进行实现和扩展，而电视就是实现接口，具体品牌的电视机是具体实现，遥控器中包含一个对电视接口的引用，通过这种方式，遥控器和电视的实现被分离，我们可以创建多个遥控器，每个遥控器控制一个品牌的电视机，它们之间独立操作，不受电视品牌的影响，可以独立变化。

简易实现

下面是实现桥接模式的基本步骤：

1. 创建实现接口

```
interface Implementation {
    void operationImpl();
}
```

以电视举例, 具有开关和切换频道的功能。

```
interface TV {
    void on();
    void off();
    void tuneChannel();
}
```

2. 创建具体实现类：实际提供服务的对象。

```java
class ConcreteImplementationA implements Implementation {
    @Override
    public void operationImpl() {
        // 具体实现A
    }
}
class ConcreteImplementationB implements Implementation {
    @Override
    public void operationImpl() {
        // 具体实现B
    }
}
```

以电视举例，创建具体实现类

```java
class ATV implements TV {
    @Override
    public void on() {
        System.out.println("A TV is ON");
    }

    @Override
    public void off() {
        System.out.println("A TV is OFF");
    }

    @Override
    public void tuneChannel() {
        System.out.println("Tuning A TV channel");
    }
}

class BTV implements TV {
    @Override
    public void on() {
        System.out.println("B TV is ON");
    }

    @Override
    public void off() {
        System.out.println("B TV is OFF");
    }

    @Override
    public void tuneChannel() {
        System.out.println("Tuning B TV channel");
    }
}
```

3. 创建抽象接口：包含一个对实现化接口的引用。

```
public abstract class Abstraction {

    protected IImplementor mImplementor;

    public Abstraction(IImplementor implementor) {
        this.mImplementor = implementor;
    }

    public void operation() {
        this.mImplementor.operationImpl();
    }

}
```

```
abstract class RemoteControl {
    // 持有一个实现化接口的引用
    protected TV tv;

    public RemoteControl(TV tv) {
        this.tv = tv;
    }

    abstract void turnOn();
    abstract void turnOff();
    abstract void changeChannel();
}
```

4. 实现抽象接口，创建`RefinedAbstaction`类

```
class RefinedAbstraction implements Abstraction {
    private Implementation implementation;

    public RefinedAbstraction(Implementation implementation) {
        this.implementation = implementation;
    }

    @Override
    public void operation() {
        // 委托给实现部分的具体类
        implementation.operationImpl();
    }
}
```

```java
class BasicRemoteControl extends RemoteControl {
    public BasicRemoteControl(TV tv) {
        super(tv);
    }

    @Override
    void turnOn() {
        tv.on();
    }

    @Override
    void turnOff() {
        tv.off();
    }

    @Override
    void changeChannel() {
        tv.tuneChannel();
    }
}
```

5. 客户端使用

```java
// 客户端代码
public class Main {
    public static void main(String[] args) {
        // 创建具体实现化对象
        Implementation implementationA = new ConcreteImplementationA();
        Implementation implementationB = new ConcreteImplementationB();

        // 使用扩充抽象化对象，将实现化对象传递进去
        Abstraction abstractionA = new RefinedAbstraction(implementationA);
        Abstraction abstractionB = new RefinedAbstraction(implementationB);

        // 调用抽象化的操作
        abstractionA.operation();
        abstractionB.operation();
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        TV aTV = new ATV();
        TV bTV = new BTV();

        RemoteControl basicRemoteForA = new BasicRemoteControl(aTV);
        RemoteControl basicRemoteForB = new BasicRemoteControl(bTV);

```

```
        basicRemoteForA.turnOn();  // A TV is ON
        basicRemoteForA.changeChannel();  // Tuning A TV channel
        basicRemoteForA.turnOff();  // A TV is OFF

        basicRemoteForB.turnOn();  // B TV is ON
        basicRemoteForB.changeChannel();  // Tuning B TV channel
        basicRemoteForB.turnOff();  // B TV is OFF
    }
}
```

使用场景

桥接模式在日常开发中使用的并不是特别多，通常在以下情况下使用：

- 当一个类存在两个独立变化的维度，而且这两个维度都需要进行扩展时，使用桥接模式可以使它们独立变化，减少耦合。
- 不希望使用继承，或继承导致类爆炸性增长

总体而言，桥接模式适用于那些有多个独立变化维度、需要灵活扩展的系统。

本题代码

```java
import java.util.Scanner;

// 步骤1：创建实现化接口
interface TV {
    void turnOn();
    void turnOff();
    void switchChannel();
}

// 步骤2：创建具体实现化类
class SonyTV implements TV {
    @Override
    public void turnOn() {
        System.out.println("Sony TV is ON");
    }

    @Override
    public void turnOff() {
        System.out.println("Sony TV is OFF");
    }

    @Override
    public void switchChannel() {
        System.out.println("Switching Sony TV channel");
    }
}
```

```java
class TCLTV implements TV {
    @Override
    public void turnOn() {
        System.out.println("TCL TV is ON");
    }

    @Override
    public void turnOff() {
        System.out.println("TCL TV is OFF");
    }

    @Override
    public void switchChannel() {
        System.out.println("Switching TCL TV channel");
    }
}

// 步骤3：创建抽象化接口
abstract class RemoteControl {
    protected TV tv;

    public RemoteControl(TV tv) {
        this.tv = tv;
    }

    abstract void performOperation();
}

// 步骤4：创建扩充抽象化类
class PowerOperation extends RemoteControl {
    public PowerOperation(TV tv) {
        super(tv);
    }

    @Override
    void performOperation() {
        tv.turnOn();
    }
}

class OffOperation extends RemoteControl {
    public OffOperation(TV tv) {
        super(tv);
    }

    @Override
    void performOperation() {
        tv.turnOff();
    }
```

```java
}

class ChannelSwitchOperation extends RemoteControl {
    public ChannelSwitchOperation(TV tv) {
        super(tv);
    }

    @Override
    void performOperation() {
        tv.switchChannel();
    }
}

// 步骤5: 客户端代码
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int N = scanner.nextInt();
        scanner.nextLine();

        for (int i = 0; i < N; i++) {
            String[] input = scanner.nextLine().split(" ");
            int brand = Integer.parseInt(input[0]);
            int operation = Integer.parseInt(input[1]);

            TV tv;
            if (brand == 0) {
                tv = new SonyTV();
            } else {
                tv = new TCLTV();
            }

            RemoteControl remoteControl;
            if (operation == 2) {
                remoteControl = new PowerOperation(tv);
            } else if (operation == 3) {
                remoteControl = new OffOperation(tv);
            } else {
                remoteControl = new ChannelSwitchOperation(tv);
            }

            remoteControl.performOperation();
        }

        scanner.close();
    }
}
```

组合模式

题目链接

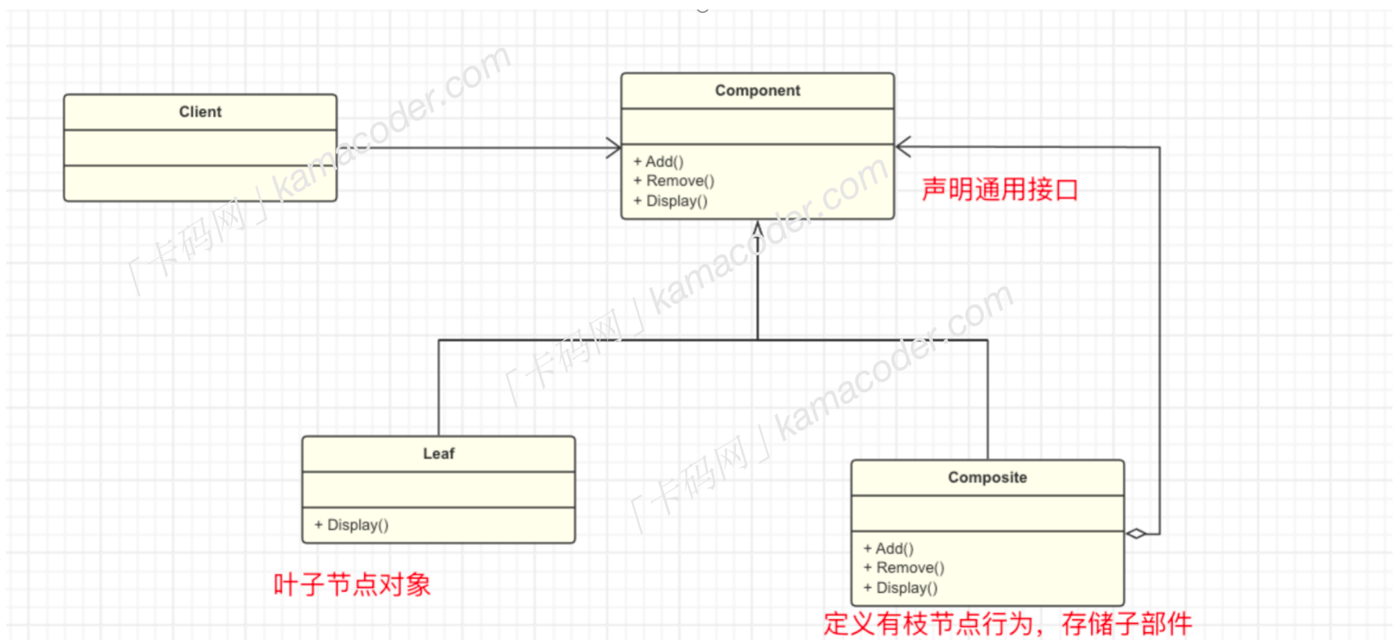
[组合模式-公司组织架构]

基本概念

组合模式是一种结构型设计模式，它**将对象组合成树状结构**来表示“部分-整体”的层次关系。组合模式使得客户端可以**统一处理单个对象和对象的组合**，而无需区分它们的具体类型。

基本结构

组合模式包括下面几个角色：



理解起来比较抽象，我们用“省份-城市”举个例子，省份中包含了多个城市，如果将之比喻成一个树形结构，城市就是叶子节点，它是省份的组成部分，而“省份”就是合成节点，可以包含其他城市，形成一个整体，省份和城市都是组件，它们都有一个共同的操作，比如获取信息。

- `Component` 组件： 组合模式的“根节点”，定义组合中所有对象的通用接口，可以是抽象类或接口。该类中定义了子类的共性内容。
- `Leaf` 叶子：实现了Component接口的叶子节点，表示组合中的叶子对象，叶子节点没有子节点。
- `Composite` 合成： 作用是存储子部件，并且在Composite中实现了对子部件的相关操作，比如添加、删除、获取子组件等。

通过组合模式，整个省份的获取信息操作可以一次性地执行，而无需关心省份中的具体城市。这样就实现了对国家省份和城市的管理和操作。

简易实现

```java
// 组件接口
interface Component {
    void operation();
}

// 叶子节点
class Leaf implements Component {
    @Override
    public void operation() {
        System.out.println("Leaf operation");
    }
}

// 组合节点：包含叶子节点的操作行为
class Composite implements Component {
    private List<Component> components = new ArrayList<>();

    public void add(Component component) {
        components.add(component);
    }

    public void remove(Component component) {
        components.remove(component);
    }

    @Override
    public void operation() {
        System.out.println("Composite operation");
        for (Component component : components) {
            component.operation();
        }
    }
}

// 客户端代码
public class Client {
    public static void main(String[] args) {
        // 创建叶子节点
        Leaf leaf = new Leaf();
        // 创建组合节点，并添加叶子节点
        Composite composite = new Composite();
        composite.add(leaf);

        composite.operation(); // 统一调用
    }
}
```

使用场景

组合模式可以使得客户端可以统一处理单个对象和组合对象，无需区分它们之间的差异，比如在图形编辑器中，图形对象可以是简单的线、圆形，也可以是复杂的组合图形，这个时候可以对组合节点添加统一的操作。

总的来说，组合模式适用于任何需要构建具有部分-整体层次结构的场景，比如组织架构管理、文件系统的文件和文件夹组织等。

本题代码

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

interface Component {
    void display(int depth);
}

class Department implements Component {
    private String name;
    private List<Component> children;

    public Department(String name) {
        this.name = name;
        this.children = new ArrayList<>();
    }

    public void add(Component component) {
        children.add(component);
    }

    @Override
    public void display(int depth) {
        StringBuilder indent = new StringBuilder();
        for (int i = 0; i < depth; i++) {
            indent.append("  ");
        }
        System.out.println(indent + name);
        for (Component component : children) {
            component.display(depth + 1);
        }
    }
}

class Employee implements Component {
    private String name;

    public Employee(String name) {
        this.name = name;
```

```
    }

    @Override
    public void display(int depth) {
        StringBuilder indent = new StringBuilder();
        for (int i = 0; i < depth; i++) {
            indent.append("  ");
        }
        System.out.println(indent + "  " + name);
    }
}

class Company {
    private String name;
    private Department root;

    public Company(String name) {
        this.name = name;
        this.root = new Department(name);
    }

    public void add(Component component) {
        root.add(component);
    }

    public void display() {
        System.out.println("Company Structure:");
        root.display(0);  // 从 1 开始，以适配指定的缩进格式
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取公司名称
        String companyName = scanner.nextLine();
        Company company = new Company(companyName);

        // 读取部门和员工数量
        int n = scanner.nextInt();
        scanner.nextLine(); // 消耗换行符

        // 读取部门和员工信息
        for (int i = 0; i < n; i++) {
            String type = scanner.next();
            String name = scanner.nextLine().trim();

            if ("D".equals(type)) {
```

```
                Department department = new Department(name);
                company.add(department);
            } else if ("E".equals(type)) {
                Employee employee = new Employee(name);
                company.add(employee);
            }
        }

        // 输出公司组织结构
        company.display();
    }
}
```

享元模式

题目链接

[享元模式-图形编辑器]

基础概念

享元模式是一种结构型设计模式，在享元模式中，对象被设计为可共享的，可以被多个上下文使用，而不必在每个上下文中都创建新的对象。

想要了解享元模式，就必须要区分什么是内部状态，什么是外部状态。

- 内部状态是指那些可以被多个对象共享的状态，它存储在享元对象内部，并且对于所有享元对象都是相同的，这部分状态通常是不变的。
- 而外部状态是享元对象依赖的、可能变化的部分。这部分状态不存储在享元对象内部，而是在使用享元对象时通过参数传递给对象。

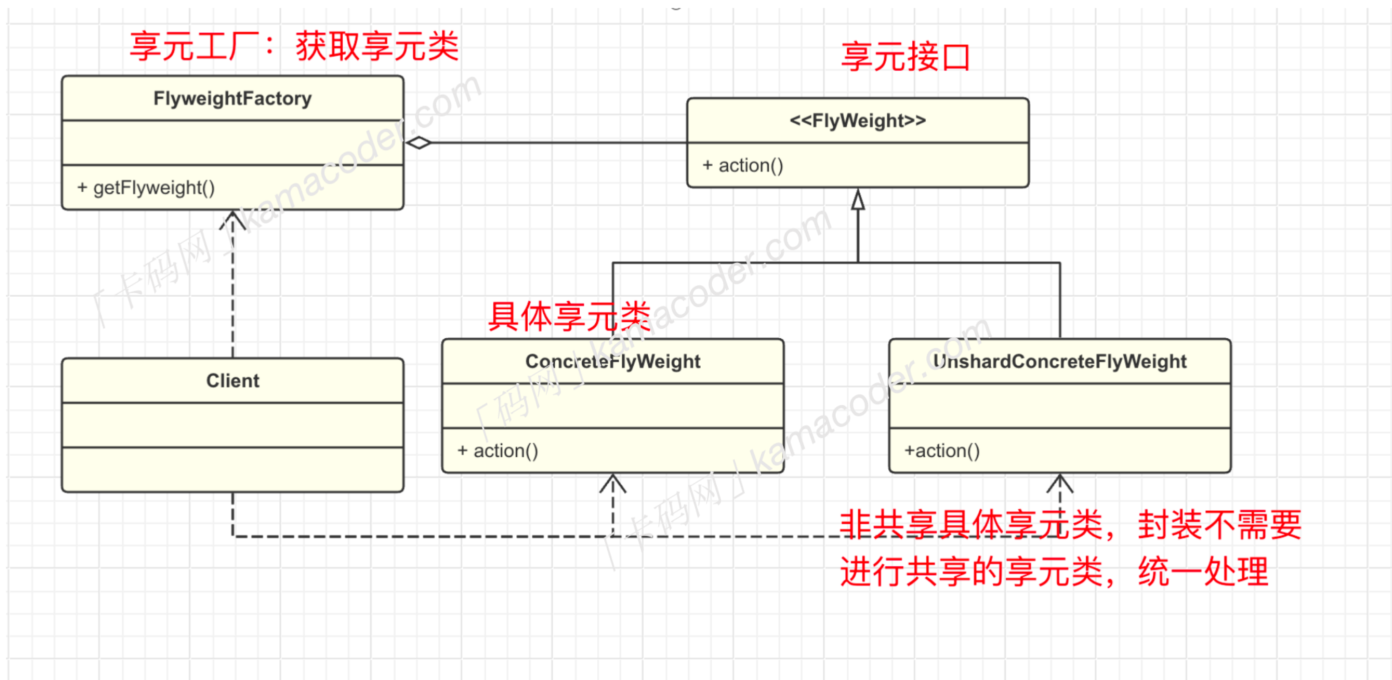
举个例子，图书馆中有很多相同的书籍，但每本书都可以被多个人借阅，图书馆里的书就是内部状态，人就是外部状态。

再举个开发中的例子，假设我们在构建一个简单的图形编辑器，用户可以在画布上绘制不同类型的图形，而图形就是所有图形对象的内部状态（不变的），而图形的坐标位置就是图形对象的外部状态（变化的）。

如果图形编辑器中有成千上万的图形对象，每个图形对象都独立创建并存储其内部状态，那么系统的内存占用可能会很大，在这种情况下，享元模式共享相同类型的图形对象，每种类型的图形对象只需创建一个共享实例，然后通过设置不同的坐标位置个性化每个对象，通过共享相同的内部状态，降低了对象的创建和内存占用成本。

基本结构

享元模式包括以下几个重要角色：



- 享元接口`Flyweight`：所有具体享元类的共享接口，通常包含对外部状态的操作。
- 具体享元类`ConcreteFlyweight`：继承`Flyweight`类或实现享元接口，包含内部状态。
- 享元工厂类`FlyweightFactory`：创建并管理享元对象，当用户请求时，提供已创建的实例或者创建一个。
- 客户端`Client`：维护外部状态，在使用享元对象时，将外部状态传递给享元对象。

简易实现

享元模式的实现通常涉及以下步骤：

1. 定义享元接口，接受外部状态作为参数并进行处理。

```
// 步骤 1：定义享元接口
interface Flyweight {
    // 操作外部状态
    void operation(String externalState);
}
```

2. 实现具体享元类，存储内部状态。

```java
// 步骤 2: 实现具体享元类
class ConcreteFlyweight implements Flyweight {
    private String intrinsicState; // 内部状态

    public ConcreteFlyweight(String intrinsicState) {
        this.intrinsicState = intrinsicState;
    }

    @Override
    public void operation(String externalState) {
        System.out.println("Intrinsic State: " + intrinsicState + ", External State: "
+ externalState);
    }
}
```

3. 创建享元工厂类，创建并管理`Flyweight`对象，当用户请求一个`Flyweight`时，享元工厂会提供一个已经创建的实例或者创建一个。

```java
class FlyweightFactory {
    private Map<String, Flyweight> flyweights = new HashMap<>();

    public Flyweight getFlyweight(String key) {
        if (!flyweights.containsKey(key)) {
            flyweights.put(key, new ConcreteFlyweight(key));
        }
        return flyweights.get(key);
    }
}
```

4. 客户端使用享元模式

```java
public class Main {
    public static void main(String[] args) {
        FlyweightFactory factory = new FlyweightFactory();

        // 获取或创建享元对象，并传递外部状态
        Flyweight flyweight1 = factory.getFlyweight("A");
        flyweight1.operation("External State 1");

        Flyweight flyweight2 = factory.getFlyweight("B");
        flyweight2.operation("External State 2");

        Flyweight flyweight3 = factory.getFlyweight("A"); // 重复使用已存在的享元对象
        flyweight3.operation("External State 3");
    }
}
```

使用场景

使用享元模式的关键在于包含大量相似对象，并且这些对象的内部状态可以共享。具体的应用场景包括文本编辑器，图形编辑器，游戏中的角色创建，这些对象的内部状态比较固定(外观，技能，形状)，但是外部状态变化比较大时，可以使用。

本题代码

```
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

enum ShapeType {
    CIRCLE, RECTANGLE, TRIANGLE
}

class Position {
    private int x;
    private int y;

    public Position(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}

interface Shape {
    void draw(Position position);
}

class ConcreteShape implements Shape {
    private ShapeType shapeType;

    public ConcreteShape(ShapeType shapeType) {
        this.shapeType = shapeType;
    }

    @Override
    public void draw(Position position) {
        System.out.println(shapeType + (isFirstTime ? " drawn" : " shared") + " at (" +
position.getX() + ", " + position.getY() + ")");
```

```java
    }

    private boolean isFirstTime = true;

    public void setFirstTime(boolean firstTime) {
        isFirstTime = firstTime;
    }
}

class ShapeFactory {
    private Map<ShapeType, Shape> shapes = new HashMap<>();

    public Shape getShape(ShapeType type) {
        if (!shapes.containsKey(type)) {
            shapes.put(type, new ConcreteShape(type));
        }
        return shapes.get(type);
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        ShapeFactory factory = new ShapeFactory();

        while (scanner.hasNext()) {
            String command = scanner.nextLine();
            processCommand(factory, command);
        }
    }

    private static void processCommand(ShapeFactory factory, String command) {
        String[] parts = command.split(" ");
        ShapeType type = ShapeType.valueOf(parts[0]);
        int x = Integer.parseInt(parts[1]);
        int y = Integer.parseInt(parts[2]);

        Shape shape = factory.getShape(type);
        shape.draw(new Position(x, y));
        ((ConcreteShape) shape).setFirstTime(false);
    }
}
```

观察者模式

题目链接

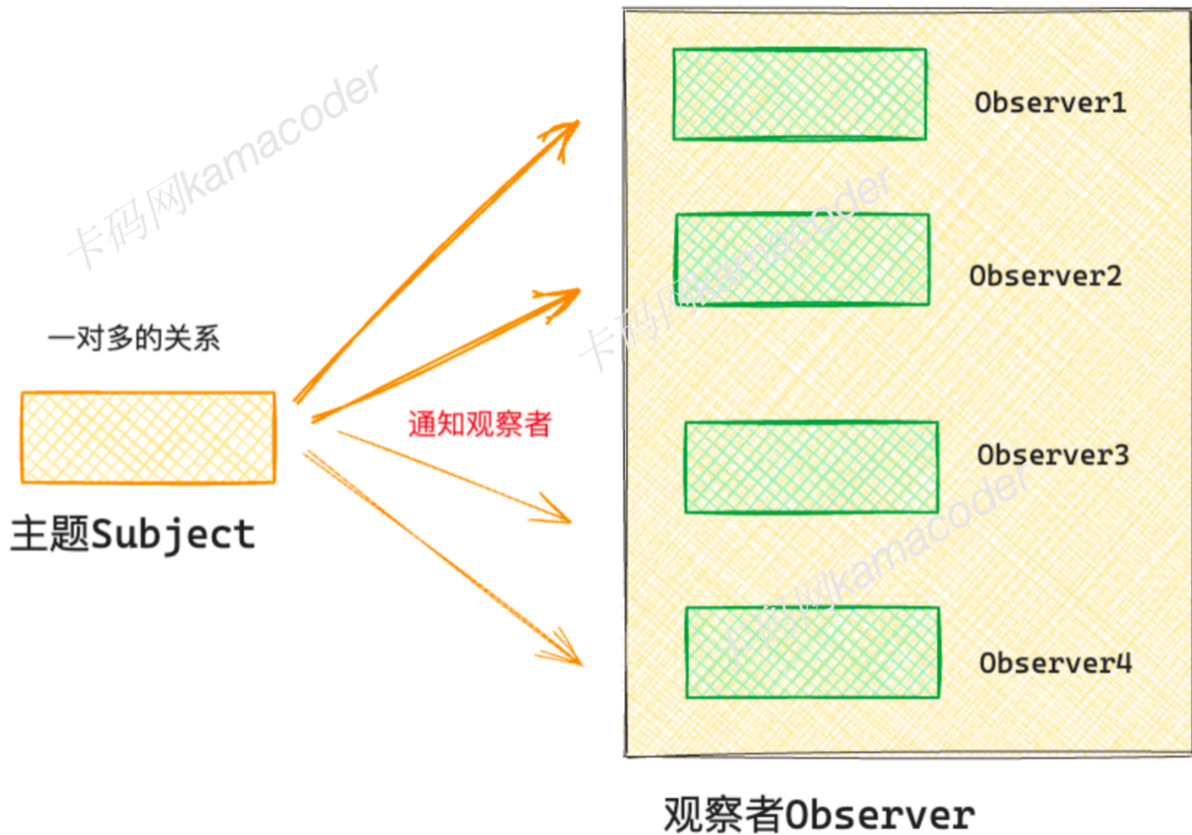
[观察者模式-时间观察者]

什么是观察者模式

观察者模式（发布-订阅模式）属于行为型模式，定义了一种一对多的依赖关系，让多个观察者对象同时监听一个主题对象，当主题对象的状态发生变化时，所有依赖于它的观察者都得到通知并被自动更新。

观察者模式依赖两个模块：

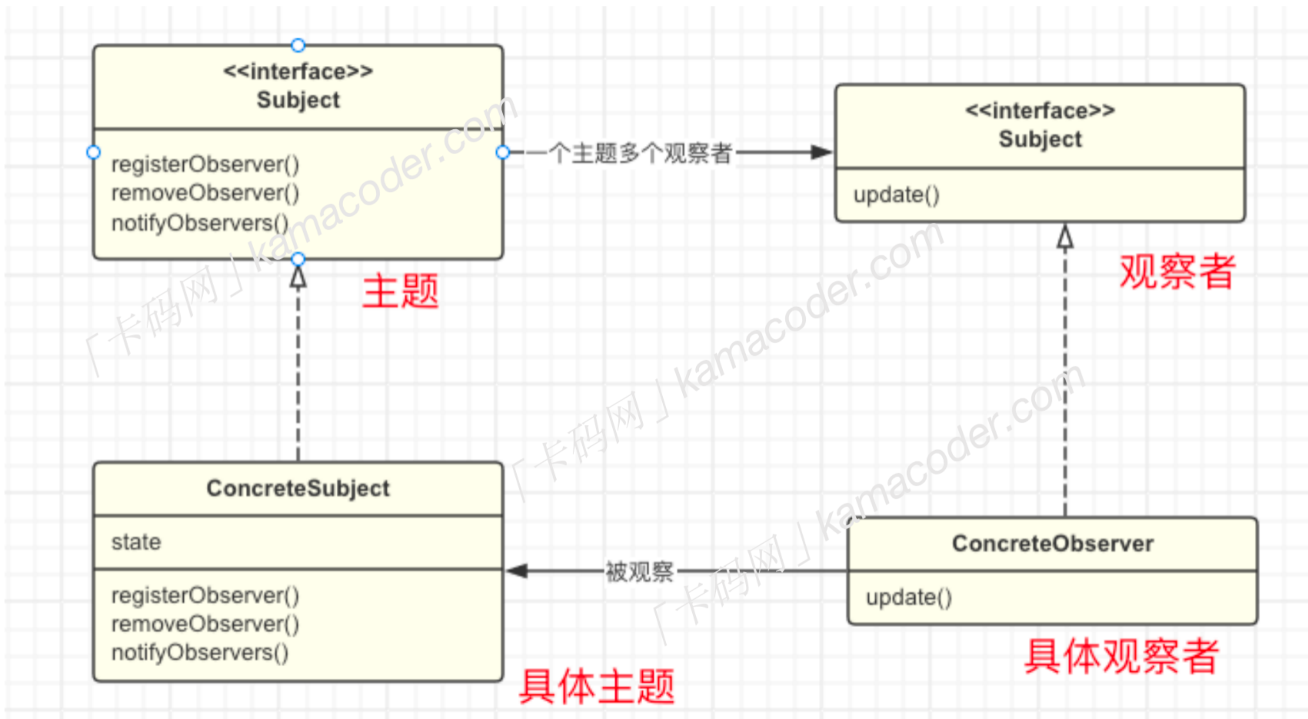
- `Subject`(主题)：也就是被观察的对象，它可以维护一组观察者，当主题本身发生改变时就会通知观察者。
- `Observer`(观察者)：观察主题的对象，当"被观察"的主题发生变化时，观察者就会得到通知并执行相应的处理。



使用观察者模式有很多好处，比如说观察者模式将主题和观察者之间的关系解耦，主题只需要关注自己的状态变化，而观察者只需要关注在主题状态变化时需要执行的操作，两者互不干扰，并且由于观察者和主题是相互独立的，可以轻松的增加和删除观察者，这样实现的系统更容易扩展和维护。

观察者模式的结构

观察者模式依赖主题和观察者，但是一般有4个组成部分：



- `主题Subject`， 一般会定义成一个接口，提供方法用于**注册、删除和通知观察者**，通常也包含一个状态，当状态发生改变时，通知所有的观察者。
- `观察者Observer`：观察者也需要实现一个接口，包含一个更新方法，在接收主题通知时执行对应的操作。
- `具体主题ConcreteSubject`：主题的具体实现, 维护一个观察者列表，包含了观察者的注册、删除和通知方法。
- `具体观察者ConcreteObserver`：观察者接口的具体实现，每个具体观察者都注册到具体主题中，当主题状态变化并通知到具体观察者，具体观察者进行处理。

观察者模式的基本实现

根据上面的类图，我们可以写出观察者模式的基本实现

```
// 主题接口 （主题）
interface Subject {
    // 注册观察者
    void registerObserver(Observer observer);
    // 移除观察者
    void removeObserver(Observer observer);
    // 通知观察者
    void notifyObservers();
}
// 观察者接口 (观察者)
interface Observer {
    // 更新方法
    void update(String message);
}

// 具体主题实现
class ConcreteSubject implements Subject {
```

```java
    // 观察者列表
    private List<Observer> observers = new ArrayList<>();
    // 状态
    private String state;

    // 注册观察者
    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }
  // 移除观察者
    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
  // 通知观察者
    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            // 观察者根据传递的信息进行处理
            observer.update(state);
        }
    }
  // 更新状态
    public void setState(String state) {
        this.state = state;
        notifyObservers();
    }
}

// 具体观察者实现
class ConcreteObserver implements Observer {
    // 更新方法
    @Override
    public void update(String message) {
    }
}
```

什么时候使用观察者模式

观察者模式特别适用于**一个对象的状态变化会影响到其他对象，并且希望这些对象在状态变化时能够自动更新的情况。** 比如说在图形用户界面中，按钮、滑动条等组件的状态变化可能需要通知其他组件更新，这使得观察者模式被广泛应用于GUI框架，比如Java的Swing框架。

此外，观察者模式在前端开发和分布式系统中也有应用，比较典型的例子是前端框架`Vue`, 当数据发生变化时，视图会自动更新。而在分布式系统中，观察者模式可以用于实现节点之间的消息通知机制，节点的状态变化将通知其他相关节点。

本题代码

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

// 观察者接口
interface Observer {
    void update(int hour);
}

// 主题接口
interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}

// 具体主题实现
class Clock implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private int hour = 0;

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(hour);
        }
    }

    public void tick() {
        hour = (hour + 1) % 24; // 模拟时间的推移
        notifyObservers();
    }
}

// 具体观察者实现
class Student implements Observer {
    private String name;
```

```java
    public Student(String name) {
        this.name = name;
    }

    @Override
    public void update(int hour) {
        System.out.println(name + " " + hour);
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取学生数量
        int N = scanner.nextInt();

        // 创建时钟
        Clock clock = new Clock();

        // 注册学生观察者
        for (int i = 0; i < N; i++) {
            String studentName = scanner.next();
            clock.registerObserver(new Student(studentName));
        }

        // 读取时钟更新次数
        int updates = scanner.nextInt();

        // 模拟时钟每隔一个小时更新一次
        for (int i = 0; i < updates; i++) {
            clock.tick();
        }
    }
}
```

策略模式

题目链接

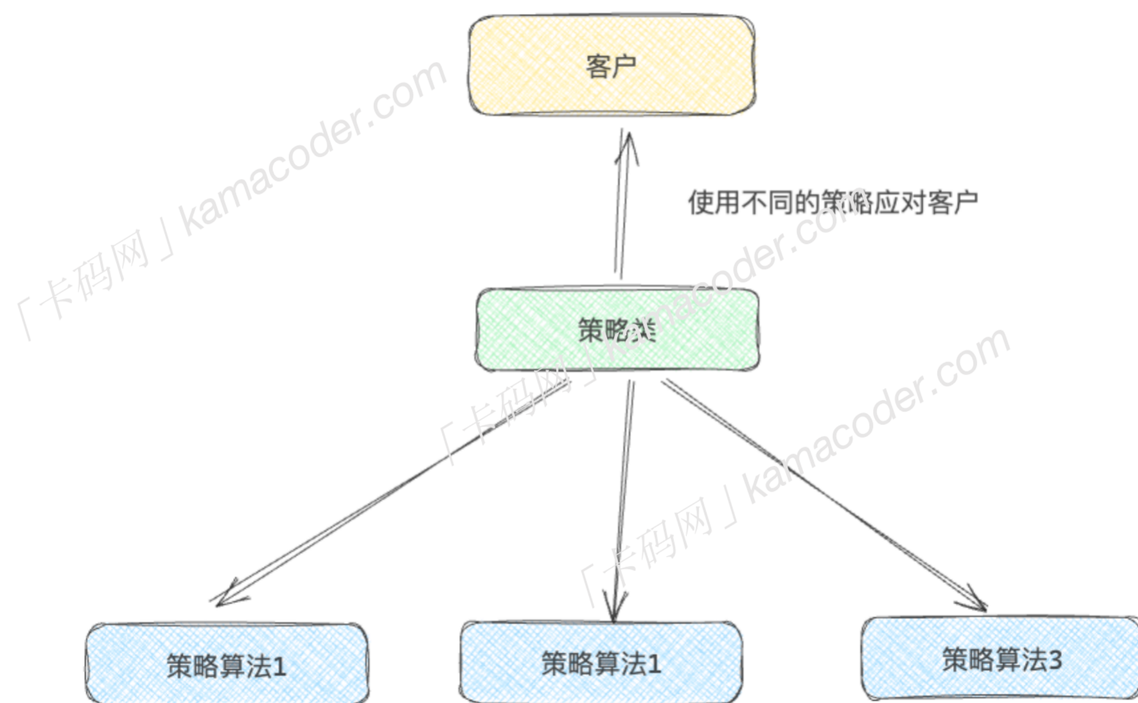
[策略模式-超市打折]

什么是策略模式

策略模式是一种行为型设计模式，它定义了一系列算法（这些算法完成的是相同的工作，只是实现不同），并将每个算法封装起来，使它们可以相互替换，而且算法的变化不会影响使用算法的客户。

举个例子，电商网站对于商品的折扣策略有不同的算法，比如新用户满减优惠，不同等级会员的打折情况不同，这种情况下会产生大量的`if-else语句`，并且如果优惠政策修改时，还需要修改原来的代码，不符合开闭原则。

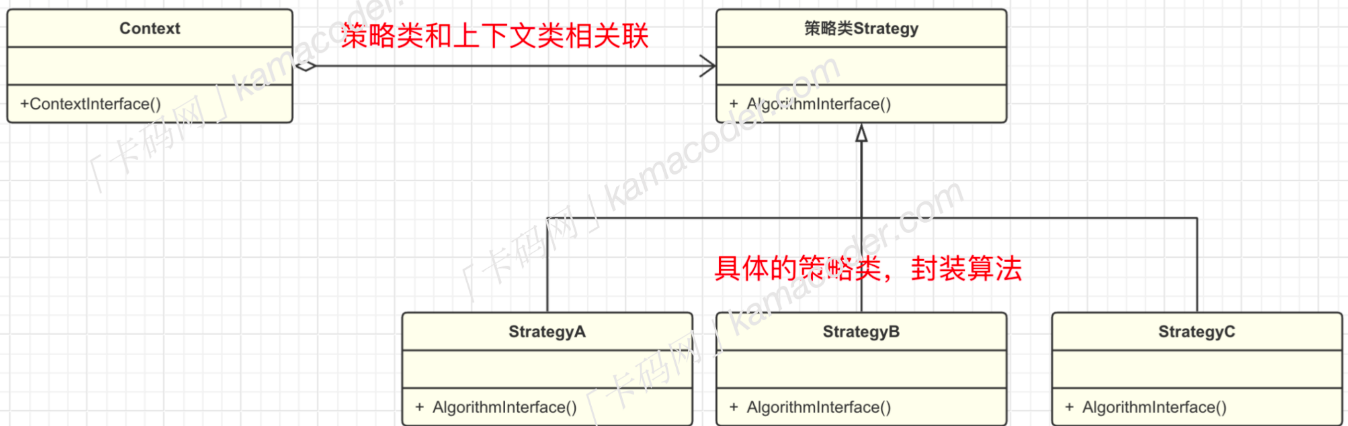
这就可以将不同的优惠算法封装成独立的类来避免大量的条件语句，如果新增优惠算法，可以添加新的策略类来实现，客户端在运行时选择不同的具体策略，而不必修改客户端代码改变优惠策略。



基本结构

策略模式包含下面几个结构：

- 策略类`Strategy`：定义所有支持的算法的公共接口。
- 具体策略类`ConcreteStrategy`：实现了策略接口，提供具体的算法实现。
- 上下文类`Context`：包含一个策略实例，并在需要时调用策略对象的方法。



简单实现

下面是一个简单的策略模式的基本实现:

```
// 1. 抽象策略抽象类
abstract class Strategy {
    // 抽象方法
    public abstract void algorithmInterface();
}

// 2. 具体策略类1
class ConcreteStrategyA extends Strategy {
    @Override
    public void algorithmInterface() {
        System.out.println("Strategy A");
        // 具体的策略1执行逻辑
    }
}

// 3. 具体策略类2
class ConcreteStrategyB extends Strategy {
    @Override
    public void algorithmInterface() {
        System.out.println("Strategy B");
        // 具体的策略2执行逻辑
    }
}

// 4. 上下文类
class Context {
    private Strategy strategy;

    // 设置具体的策略
```

```java
     public Context(Strategy strategy) {
        this.strategy = strategy;
     }

    // 执行策略
    public void contextInterface() {
        strategy.algorithmlnterface();
    }
}

// 5. 客户端代码
public class Main{
    public static void main(String[] args) {
        // 创建上下文对象，并设置具体的策略
        Context contextA = new Context(new ConcreteStrategyA());
        // 执行策略
        contextA.contextInterface();

        Context contextB = new Context(new ConcreteStrategyB());
        contextB.contextInterface();u
    }
}
```

使用场景

那什么时候可以考虑使用策略模式呢?

- 当一个系统根据业务场景需要动态地在几种算法中选择一种时，可以使用策略模式。例如，根据用户的行为选择不同的计费策略。
- 当代码中存在大量条件判断，条件判断的区别仅仅在于行为，也可以通过策略模式来消除这些条件语句。

在已有的工具库中，Java 标准库中的 `Comparator` 接口就使用了策略模式，通过实现这个接口，可以创建不同的比较器（指定不同的排序策略）来满足不同的排序需求。

本题代码

```java
import java.util.Scanner;

// 抽象购物优惠策略接口
interface DiscountStrategy {
    int applyDiscount(int originalPrice);
}

// 九折优惠策略
class DiscountStrategy1 implements DiscountStrategy {
    @Override
    public int applyDiscount(int originalPrice) {
        return (int) Math.round(originalPrice * 0.9);
    }
```

```
}

// 满减优惠策略
class DiscountStrategy2 implements DiscountStrategy {
    private int[] thresholds = {100, 150, 200, 300};
    private int[] discounts = {5, 15, 25, 40};

    @Override
    public int applyDiscount(int originalPrice) {
        for (int i = thresholds.length - 1; i >= 0; i--) {
            if (originalPrice >= thresholds[i]) {
                return originalPrice - discounts[i];
            }
        }
        return originalPrice;
    }
}

// 上下文类
class DiscountContext {
    private DiscountStrategy discountStrategy;

    public void setDiscountStrategy(DiscountStrategy discountStrategy) {
        this.discountStrategy = discountStrategy;
    }

    public int applyDiscount(int originalPrice) {
        return discountStrategy.applyDiscount(originalPrice);
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取需要计算优惠的次数
        int N = Integer.parseInt(scanner.nextLine());

        for (int i = 0; i < N; i++) {
            // 读取商品价格和优惠策略
            String[] input = scanner.nextLine().split(" ");
            int M = Integer.parseInt(input[0]);
            int strategyType = Integer.parseInt(input[1]);

            // 根据优惠策略设置相应的打折策略
            DiscountStrategy discountStrategy;
            switch (strategyType) {
                case 1:
                    discountStrategy = new DiscountStrategy1();
```

```
                break;
            case 2:
                discountStrategy = new DiscountStrategy2();
                break;
            default:
                // 处理未知策略类型
                System.out.println("Unknown strategy type");
                return;
        }

        // 设置打折策略
        DiscountContext context = new DiscountContext();
        context.setDiscountStrategy(discountStrategy);

        // 应用打折策略并输出优惠后的价格
        int discountedPrice = context.applyDiscount(M);
        System.out.println(discountedPrice);
    }
  }
}
```

命令模式

题目链接

命令模式-自助点餐机

基本概念

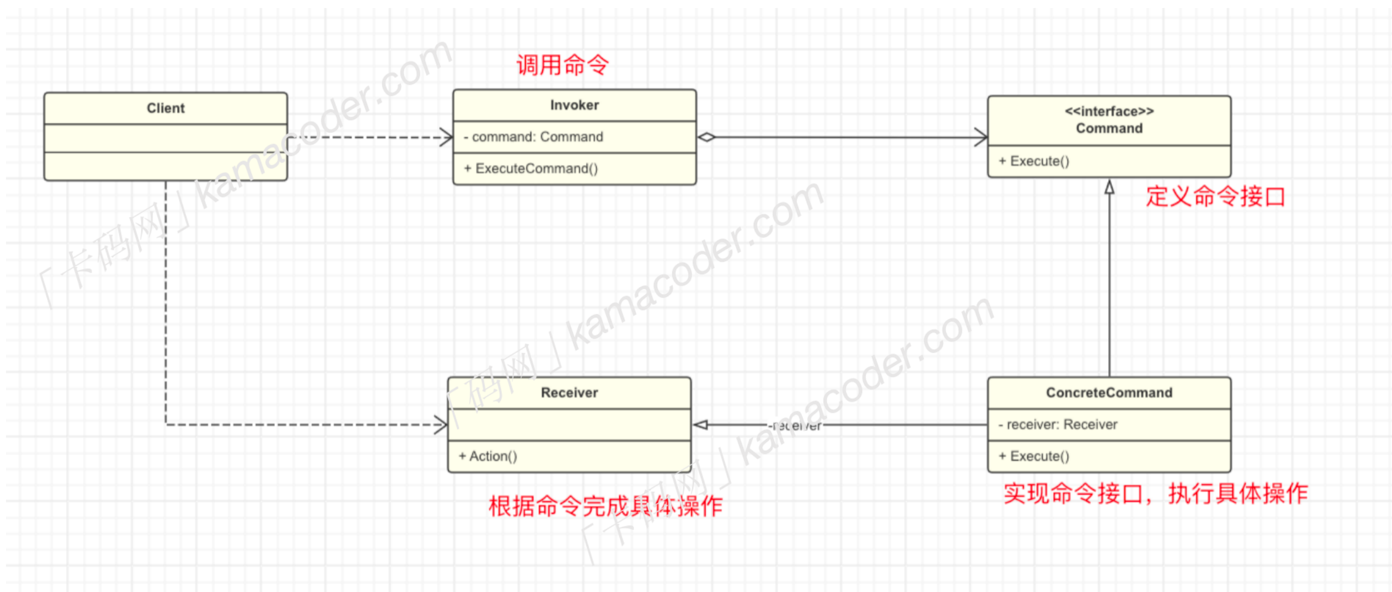
命令模式是一种行为型设计模式，其允许**将请求封装成一个对象**(命令对象，包含执行操作所需的所有信息)，**并将命令对象按照一定的顺序存储在队列中，然后再逐一调用执行，这些命令也可以支持反向操作，进行撤销和重做。**

这样一来，发送者只需要触发命令就可以完成操作，不需要知道接受者的具体操作，从而实现两者间的解耦。

举个现实中的应用场景，遥控器可以控制不同的设备，在命令模式中，可以假定每个按钮都是一个命令对象，包含执行特定操作的命令，不同设备对同一命令的具体操作也不同，这样就可以方便的添加设备和命令对象。

基本结构

命令模式包含以下几个基本角色：



- 命令接口`Command`：接口或者抽象类，定义执行操作的接口。
- 具体命令类`ConcreteCommand`: 实现命令接口，执行具体操作，在调用`execute`方法时使“接收者对象”根据命令完成具体的任务，比如遥控器中的“开机”，“关机”命令。
- 接收者类`Receiver`: 接受并执行命令的对象，可以是任何对象，遥控器可以控制空调，也可以控制电视机，电视机和空调负责执行具体操作，是接收者。
- 调用者类`Invoker`：发起请求的对象，有一个将命令作为参数传递的方法。它不关心命令的具体实现，只负责调用命令对象的 `execute()` 方法来传递请求，在本例中，控制遥控器的“人”就是调用者。
- 客户端：创建具体的命令对象和接收者对象，然后将它们组装起来。

简易实现

1. 定义执行操作的接口：包含一个`execute`方法。有的时候还会包括`unExecute`方法，表示撤销命令。

```
public interface Command {
    void execute();
}
```

2. 实现命令接口，执行具体的操作。

```
public class ConcreteCommand implements Command {
    // 接收者对象
    private Receiver receiver;

    public ConcreteCommand(Receiver receiver) {
        this.receiver = receiver;
    }

    @Override
    public void execute() {
        // 调用接收者相应的操作
        receiver.action();
    }
```

```
}
```

3. 定义接受者类，知道如何实施与执行一个请求相关的操作。

```
public class Receiver {
    public void action() {
        // 执行操作
    }
}
```

4. 定义调用者类，调用命令对象执行请求。

```
public class Invoker {
    private Command command;

    public Invoker(Command command) {
        this.command = command;
    }

    public void executeCommand() {
        command.execute();
    }
}
```

调用者类中可以维护一个命令队列或者“撤销栈”，以支持批处理和撤销命令。

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

// 调用者类：命令队列和撤销请求
class Invoker {
    private Queue<Command> commandQueue; // 命令队列
    private Stack<Command> undoStack;    // 撤销栈

    public Invoker() {
        this.commandQueue = new LinkedList<>();
        this.undoStack = new Stack<>();
    }

    // 设置命令并执行
    public void setAndExecuteCommand(Command command) {
        command.execute();
        commandQueue.offer(command);
        undoStack.push(command);
    }

    // 撤销上一个命令
```

```
    public void undoLastCommand() {
        if (!undoStack.isEmpty()) {
            Command lastCommand = undoStack.pop();
            lastCommand.undo(); // 需要命令类实现 undo 方法
            commandQueue.remove(lastCommand);
        } else {
            System.out.println("No command to undo.");
        }
    }

    // 执行命令队列中的所有命令
    public void executeCommandsInQueue() {
        for (Command command : commandQueue) {
            command.execute();
        }
    }
}
```

5. 客户端使用，创建具体的命令对象和接收者对象，然后进行组装。

```
public class Main {
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        Command command = new ConcreteCommand(receiver);
        Invoker invoker = new Invoker(command);

        invoker.executeCommand();
    }
}
```

优缺点和使用场景

命令模式在需要将请求封装成对象、支持撤销和重做、设计命令队列等情况下，都是一个有效的设计模式。

- **撤销操作**：需要支持撤销操作，命令模式可以存储历史命令，轻松实现撤销功能。
- **队列请求**：命令模式可以将请求排队，形成一个命令队列，依次执行命令。
- **可扩展性**：可以很容易地添加新的命令类和接收者类，而不影响现有的代码。新增命令不需要修改现有代码，符合开闭原则。

但是对于每个命令，都会有一个具体命令类，这可能导致类的数量急剧增加，增加了系统的复杂性。

命令模式同样有着很多现实场景的应用，比如Git中的很多操作，如提交（commit）、合并（merge）等，都可以看作是命令模式的应用，用户通过执行相应的命令来操作版本库。Java的GUI编程中，很多事件处理机制也都使用了命令模式。例如，每个按钮都有一个关联的 `Action`，它代表一个命令，按钮的点击触发 `Action` 的执行。

本题代码

```java
import java.util.Scanner;

// 命令接口
interface Command {
    void execute();
}

// 具体命令类 - 点餐命令
class OrderCommand implements Command {
    private String drinkName;
    private DrinkMaker receiver;

    public OrderCommand(String drinkName, DrinkMaker receiver) {
        this.drinkName = drinkName;
        this.receiver = receiver;
    }

    @Override
    public void execute() {
        receiver.makeDrink(drinkName);
    }
}

// 接收者类 - 制作饮品
class DrinkMaker {
    public void makeDrink(String drinkName) {
        System.out.println(drinkName + " is ready!");
    }
}

// 调用者类 - 点餐机
class OrderMachine {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void executeOrder() {
        command.execute();
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
```

```java
        // 创建接收者和命令对象
        DrinkMaker drinkMaker = new DrinkMaker();

        // 读取命令数量
        int n = scanner.nextInt();
        scanner.nextLine(); // 消耗掉换行符

        while (n-- > 0) {
            // 读取命令
            String drinkName = scanner.next();

            // 创建命令对象
            Command command = new OrderCommand(drinkName, drinkMaker);

            // 执行命令
            OrderMachine orderMachine = new OrderMachine();
            orderMachine.setCommand(command);
            orderMachine.executeOrder();
        }
        scanner.close();
    }
}
```

中介者模式

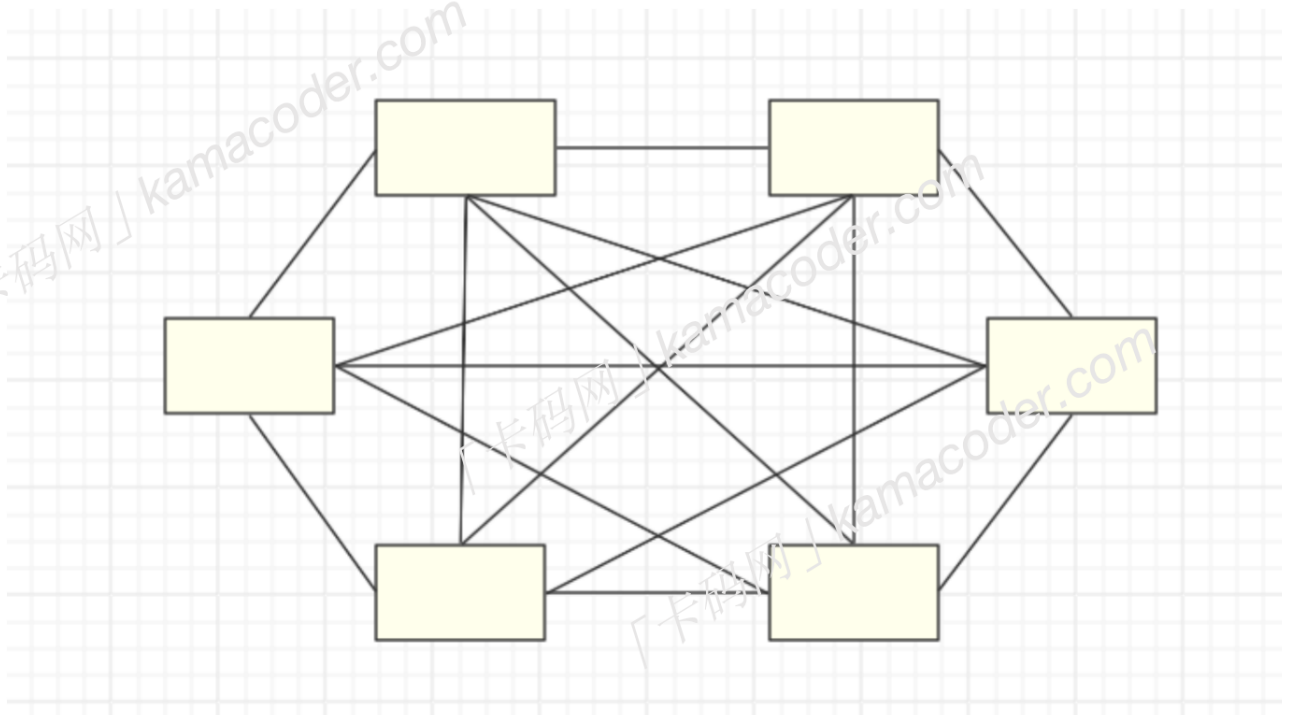
题目链接

[中介者模式-简易聊天室]

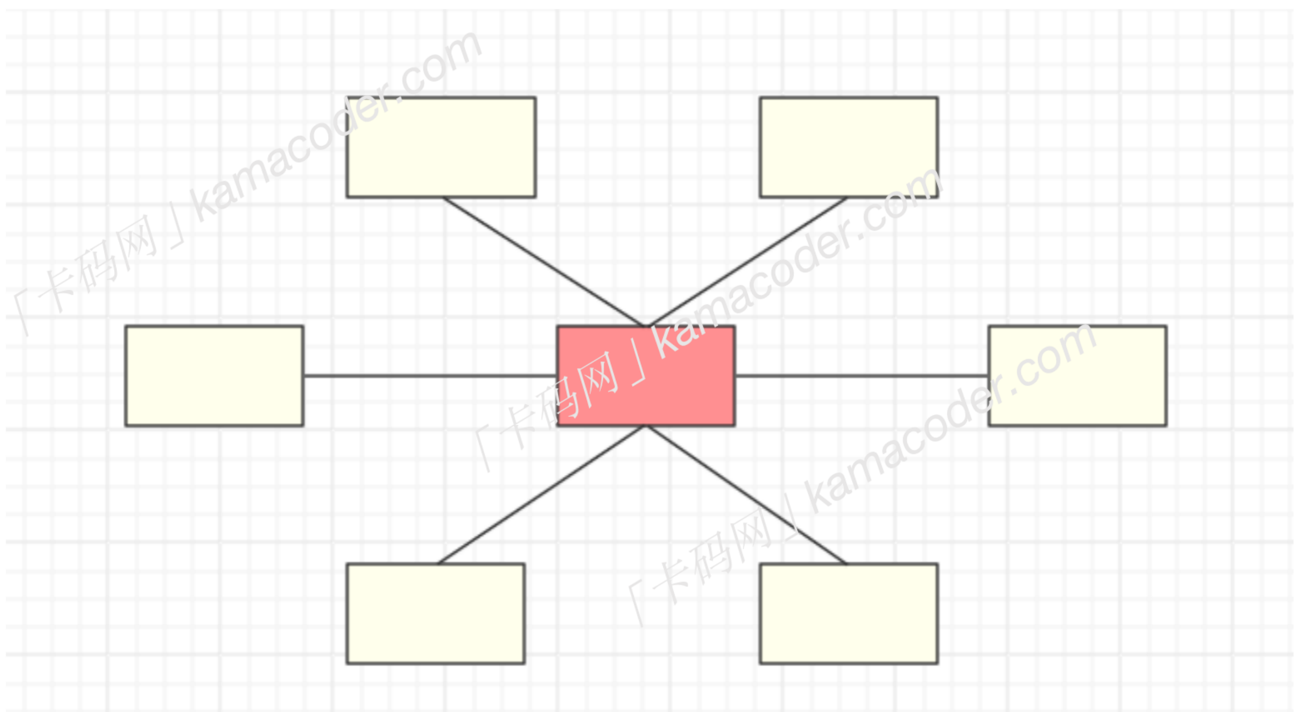
基本概念

中介者模式（Mediator Pattern）也被称为调停者模式，是一种行为型设计模式，它通过一个中介对象来封装一组对象之间的交互，从而使这些对象不需要直接相互引用。这样可以降低对象之间的耦合度，使系统更容易维护和扩展。

当一个系统中的对象有很多且多个对象之间有复杂的相互依赖关系时，其结构图可能是下面这样的。

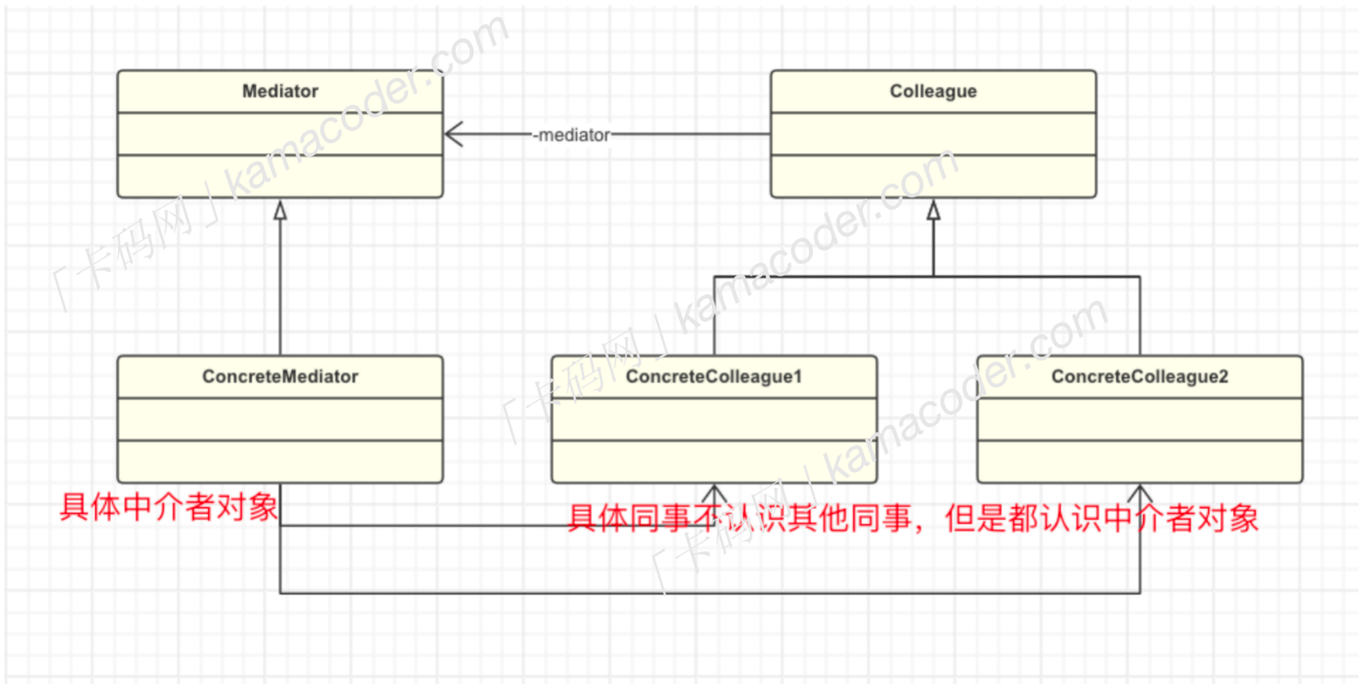


这种依赖关系很难理清，这时我们可以引入一个中介者对象来进行协调和交互。中介者模式可以使得系统的网状结构变成以中介者为中心的星形结构，每个具体对象不再通过直接的联系与另一个对象发生相互作用，而是通过“中介者”对象与另一个对象发生相互作用。



基本结构

中介者模式包括以下几个重要角色：



- **抽象中介者（Mediator）**： 定义中介者的接口，用于各个具体同事对象之间的通信。
- **具体中介者（Concrete Mediator）**： 实现抽象中介者接口，负责协调各个具体同事对象的交互关系，它需要知道所有具体同事类，并从具体同事接收消息，向具体同事对象发出命令。
- **抽象同事类（Colleague）**： 定义同事类的接口，维护一个对中介者对象的引用，用于通信。
- **具体同事类（Concrete Colleague）**： 实现抽象同事类接口，每个具体同事类只知道自己的行为，而不了解其他同事类的情况，因为它们都需要与中介者通信，通过中介者协调与其他同事对象的交互。

简易实现

```
// 抽象中介者
public abstract class Mediator {
    void register(Colleague colleague);
    // 定义一个抽象的发送消息方法
    public abstract void send(String message, Player player);

}

// 具体中介者
public class ConcreteMediator extends Mediator {
    private List<Colleague> colleagues = new ArrayList<>();

    public void register((Colleague colleague) {
        colleagues.add(colleague);
    }

    @Override
```

```java
    public void send(String message, Colleague colleague) {
        for (Colleague c : colleagues) {
            // 排除发送消息的同事对象
            if (c != colleague) {
                c.receive(message);
            }
        }
    }
}

// 同事对象
abstract class Colleague {
    protected Mediator mediator;

    public Colleague(Mediator mediator) {
        this.mediator = mediator;
    }

    // 发送消息
    public abstract void send(String message);

    // 接收消息
    public abstract void receive(String message);
}

// 具体同事对象1
class ConcreteColleague1 extends Colleague {
    public ConcreteColleague1(Mediator mediator) {
        super(mediator);
    }

    @Override
    public void send(String message) {
        mediator.send(message, this);
    }

    @Override
    public void receive(String message) {
        System.out.println("ConcreteColleague1 received: " + message);
    }
}

// 具体同事对象2
class ConcreteColleague2 extends Colleague {
    public ConcreteColleague2(Mediator mediator) {
        super(mediator);
    }

    @Override
```

```
    public void send(String message) {
        mediator.send(message, this);
    }

    @Override
    public void receive(String message) {
        System.out.println("ConcreteColleague2 received: " + message);
    }
}

// 客户端
public class Main{
    public static void main(String[] args) {
        // 创建中介者
        Mediator mediator = new ConcreteMediator();

        // 创建同事对象
        Colleague colleague1 = new ConcreteColleague1(mediator);
        Colleague colleague2 = new ConcreteColleague2(mediator);

        // 注册同事对象到中介者
        mediator.register(colleague1);
        mediator.register(colleague2);

        // 同事对象之间发送消息
        colleague1.send("Hello from Colleague1!");
        colleague2.send("Hi from Colleague2!");
    }
}
```

使用场景

中介者模式使得同事对象不需要知道彼此的细节，只需要与中介者进行通信，简化了系统的复杂度，也降低了各对象之间的耦合度，但是这也会使得中介者对象变得过于庞大和复杂，如果中介者对象出现问题，整个系统可能会受到影响。

中介者模式适用于当系统对象之间存在复杂的交互关系或者系统需要在不同对象之间进行灵活的通信时使用，可以使得问题简化，

本题代码

```
import java.util.*;

// 抽象中介者
interface ChatRoomMediator {
    void sendMessage(String sender, String message);
    void addUser(ChatUser user);
    Map<String, ChatUser> getUsers();
}
```

```java
// 具体中介者
class ChatRoomMediatorImpl implements ChatRoomMediator {
    private Map<String, ChatUser> users = new LinkedHashMap<>();

    @Override
    public void sendMessage(String sender, String message) {
        for (ChatUser user : users.values()) {
            if (!user.getName().equals(sender)) {
                user.receiveMessage(sender, message);
            }
        }
    }

    @Override
    public void addUser(ChatUser user) {
        users.put(user.getName(), user);
    }

    @Override
    public Map<String, ChatUser> getUsers() {
        return users;
    }
}

// 抽象同事类
abstract class ChatUser {
    private String name;
    private ChatRoomMediator mediator;
    private List<String> receivedMessages = new ArrayList<>();

    public ChatUser(String name, ChatRoomMediator mediator) {
        this.name = name;
        this.mediator = mediator;
        mediator.addUser(this);
    }

    public String getName() {
        return name;
    }

    public void sendMessage(String message) {
        mediator.sendMessage(name, message);
    }

    public abstract void receiveMessage(String sender, String message);

    public List<String> getReceivedMessages() {
        return receivedMessages;
```

```java
    }

    protected void addReceivedMessage(String message) {
        receivedMessages.add(message);
    }
}

// 具体同事类
class ConcreteChatUser extends ChatUser {
    public ConcreteChatUser(String name, ChatRoomMediator mediator) {
        super(name, mediator);
    }

    @Override
    public void receiveMessage(String sender, String message) {
        String receivedMessage = getName() + " received: " + message;
        addReceivedMessage(receivedMessage);
        System.out.println(receivedMessage);
    }
}

// 客户端
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int N = scanner.nextInt();
        List<String> userNames = new ArrayList<>();
        for (int i = 0; i < N; i++) {
            userNames.add(scanner.next());
        }

        ChatRoomMediator mediator = new ChatRoomMediatorImpl();

        // 创建用户对象
        for (String userName : userNames) {
            new ConcreteChatUser(userName, mediator);
        }

        // 发送消息并输出
        while (scanner.hasNext()) {
            String sender = scanner.next();
            String message = scanner.next();

            ChatUser user = mediator.getUsers().get(sender);
            if (user != null) {
                user.sendMessage(message);
            }
        }
```

```
        scanner.close();
    }
}
```

扩展：和代理模式的区别

中介者模式（Mediator Pattern）和代理模式（Proxy Pattern) 在某些表述上有些类似，但是他们是完全不同的两个设计模式，中介者模式的目的是降低系统中各个对象之间的直接耦合，通过引入一个中介者对象，使对象之间的通信集中在中介者上。而在代理模式中，客户端通过代理与目标对象进行通信。代理可以在调用目标对象的方法前后进行一些额外的操作，其目的是控制对对象的访问，它们分别解决了不同类型的问题。

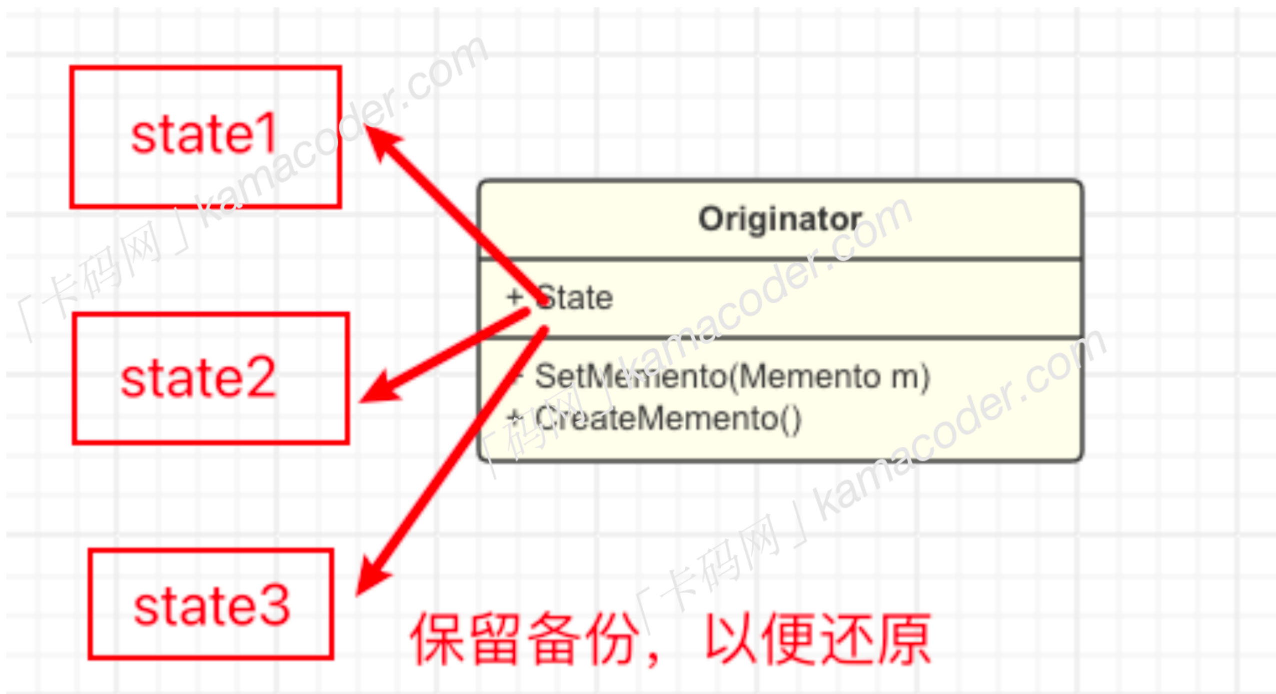
备忘录模式

题目链接

[备忘录模式-redo计数器应用]()

基本概念

备忘录模式（Memento Pattern）是一种行为型设计模式，它允许在**不暴露对象实现的情况下捕获对象的内部状态并在对象之外保存这个状态**，以便稍后可以将其还原到先前的状态。



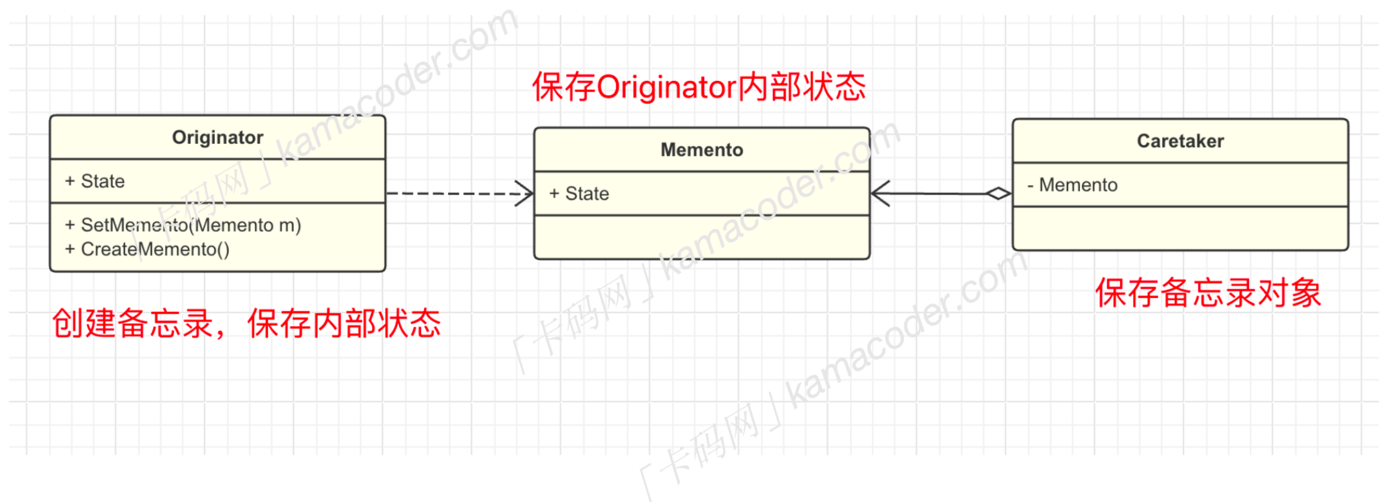
基本结构

备忘录模式包括以下几个重要角色：

- 发起人`Originator`： 需要还原状态的那个对象，负责创建一个【备忘录】，并使用备忘录记录当前时刻的内部状态。
- 备忘录`Memento`：存储发起人对象的内部状态，它可以包含发起人的部分或全部状态信息，但是对外部是不可见的，只有发起人能够访问备忘录对象的状态。

> 备忘录有两个接口，发起人能够通过宽接口访问数据，管理者只能看到窄接口，并将备忘录传递给其他对象。

- 管理者`Caretaker`：负责存储备忘录对象，但并不了解其内部结构，管理者可以存储多个备忘录对象。
- 客户端：在需要恢复状态时，客户端可以从管理者那里获取备忘录对象，并将其传递给发起人进行状态的恢复。



基本实现

1. 创建发起人类：可以创建备忘录对象

```
class Originator {
    private String state;

    public void setState(String state) {
        this.state = state;
    }
    public String getState() {
        return state;
    }
  // 创建备忘录对象
    public Memento createMemento() {
        return new Memento(state);
    }
  // 通过备忘录对象恢复状态
    public void restoreFromMemento(Memento memento) {
```

```
        state = memento.getState();
    }
}
```

2. 创建备忘录类：保存发起人对象的状态

```
class Memento {

    private String state;
  // 保存发起人的状态
    public Memento(String state) {
        this.state = state;
    }

    public String getState() {
        return state;
    }
}
```

3. 创建管理者：维护一组备忘录对象

```
class Caretaker {
    private List<Memento> mementos = new ArrayList<>();

    public void addMemento(Memento memento) {
        mementos.add(memento);
    }

    public Memento getMemento(int index) {
        return mementos.get(index);
    }
}
```

4. 客户端使用备忘录模式

```
public class Main {
    public static void main(String[] args) {
        // 创建发起人对象
        Originator originator = new Originator();
        originator.setState("State 1");

        // 创建管理者对象
        Caretaker caretaker = new Caretaker();

        // 保存当前状态
        caretaker.addMemento(originator.createMemento());

        // 修改状态
```

```
        originator.setState("State 2");

        // 再次保存当前状态
        caretaker.addMemento(originator.createMemento());

        // 恢复到先前状态
        originator.restoreFromMemento(caretaker.getMemento(0));

        System.out.println("Current State: " + originator.getState());
    }
}
```

使用场景

备忘录模式在保证了对象内部状态的封装和私有性前提下可以轻松地添加新的备忘录和发起人，实现“备份”，不过备份对象往往会消耗较多的内存，资源消耗增加。

备忘录模式常常用来实现撤销和重做功能，比如在Java Swing GUI编程中，`javax.swing.undo`包中的撤销（undo）和重做（redo）机制使用了备忘录模式。`UndoManager`和`UndoableEdit`接口是与备忘录模式相关的主要类和接口。

本题代码

```
import java.util.Scanner;
import java.util.Stack;

// 备忘录
class Memento {
    private int value;

    public Memento(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

// 发起人（Originator）
class Counter {
    private int value;
    private Stack<Memento> undoStack = new Stack<>();
    private Stack<Memento> redoStack = new Stack<>();

    public void increment() {
        redoStack.clear();
        undoStack.push(new Memento(value));
        value++;
```

```java
    }

    public void decrement() {
        redoStack.clear();
        undoStack.push(new Memento(value));
        value--;
    }

    public void undo() {
        if (!undoStack.isEmpty()) {
            redoStack.push(new Memento(value));
            value = undoStack.pop().getValue();
        }
    }

    public void redo() {
        if (!redoStack.isEmpty()) {
            undoStack.push(new Memento(value));
            value = redoStack.pop().getValue();
        }
    }

    public int getValue() {
        return value;
    }
}

// 客户端
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Counter counter = new Counter();

        // 处理计数器应用的输入
        while (scanner.hasNext()) {
            String operation = scanner.next();
            switch (operation) {
                case "Increment":
                    counter.increment();
                    break;
                case "Decrement":
                    counter.decrement();
                    break;
                case "Undo":
                    counter.undo();
                    break;
                case "Redo":
                    counter.redo();
                    break;
```

```
            }

            // 输出当前计数器的值
            System.out.println(counter.getValue());
        }

        scanner.close();
    }
}
```

模板方法模式

题目链接

[模板方法模式-咖啡馆]

基本概念

模板方法模式（Template Method Pattern）是一种行为型设计模式, 它定义了一个算法的骨架，将**一些步骤的实现延迟到子类**。模板方法模式使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。【引用自大话设计第10章】

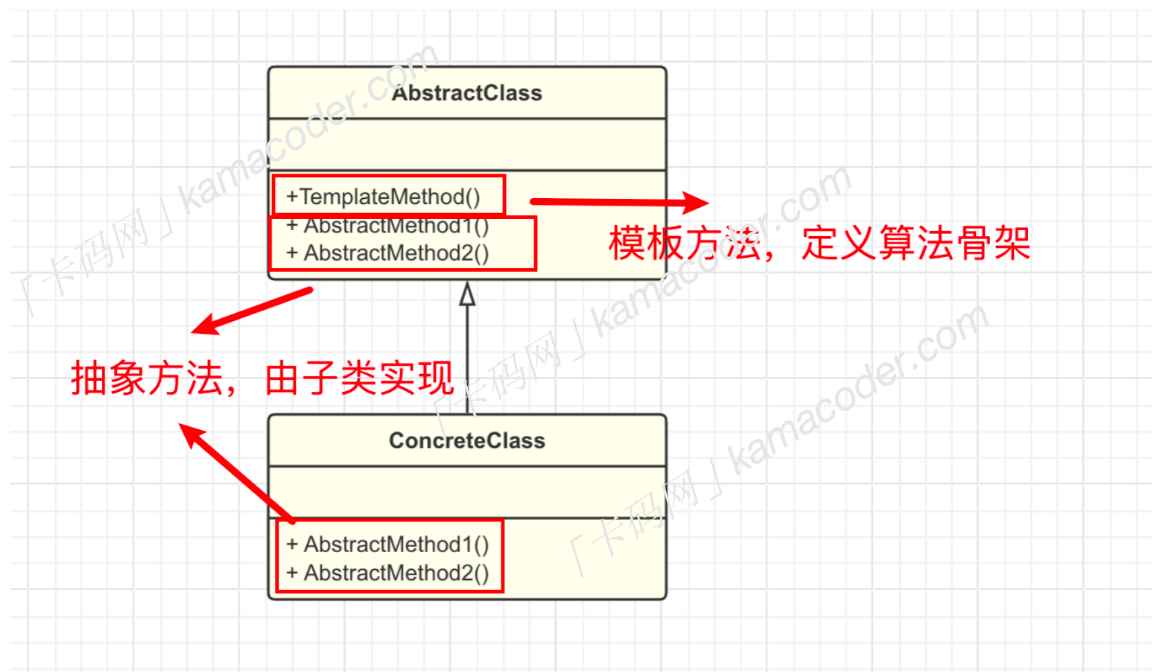
举个简单的例子，做一道菜通常都需要包含至少三步：

- 准备食材
- 亨饪过程
- 上菜

不同菜品的亨饪过程是不一样的，但是我们可以先定义一个"骨架"，包含这三个步骤，亨饪过程的过程放到具体的炒菜类中去实现，这样，无论炒什么菜，都可以沿用相同的炒菜算法，只需在子类中实现具体的炒菜步骤，从而提高了代码的复用性。

基本结构

模板方法模式的基本结构包含以下两个角色：



- 模板类`AbstractClass`：由一个模板方法和若干个基本方法构成，模板方法定义了逻辑的骨架，按照顺序调用包含的基本方法，基本方法通常是一些**抽象方法，这些方法由子类去实现**。基本方法还包含一些具体方法，它们是算法的一部分但已经有默认实现，在具体子类中可以继承或者重写。
- 具体类`ConcreteClass`：继承自模板类，实现了在模板类中定义的抽象方法，以完成算法中特定步骤的具体实现。

简易实现

模板方法模式的简单示例如下：

1. 定义模板类，包含模板方法，定义了算法的骨架, 一般都加上`final`关键字，避免子类重写。

```
// 模板类
abstract class AbstractClass {
    // 模板方法，定义了算法的骨架
    public final void templateMethod() {
        step1();
        step2();
        step3();
    }

    // 抽象方法，由子类实现
    protected abstract void step1();
    protected abstract void step2();
    protected abstract void step3();
}
```

2. 定义具体类, 实现模板类中的抽象方法

```
// 具体类
class ConcreteClass extends AbstractClass {
```

```
    @Override
    protected void step1() {
        System.out.println("Step 1 ");
    }

    @Override
    protected void step2() {
        System.out.println("Step 2 ");
    }

    @Override
    protected void step3() {
        System.out.println("Step 3");
    }
}
```

3. 客户端实现

```
public class Main {
    public static void main(String[] args) {
        AbstractClass concreteTemplate = new ConcreteClass();
        // 触发整个算法的执行
        concreteTemplate.templateMethod();
    }
}
```

应用场景

模板方法模式将算法的不变部分被封装在模板方法中，而可变部分算法由子类继承实现，这样做可以很好的提高代码的复用性，但是当算法的框架发生变化时，可能需要修改模板类，这也会影响到所有的子类。

总体来说，当算法的整体步骤很固定，但是个别步骤在更详细的层次上的实现可能不同时，通常考虑模板方法模式来处理。在已有的工具和库中， Spring框架中的`JdbcTemplate`类使用了模板方法模式，其中定义了一些执行数据库操作的模板方法，具体的数据库操作由回调函数提供。而在Java的JDK源码中， `AbstractList` 类也使用了模板方法模式，它提供了一些通用的方法，其中包括一些模板方法。具体的列表操作由子类实现。

本题代码

```
import java.util.Scanner;

// 抽象类
abstract class CoffeeMakerTemplate {
    private String coffeeName; // 添加咖啡名称字段

    // 构造函数，接受咖啡名称参数
    public CoffeeMakerTemplate(String coffeeName) {
        this.coffeeName = coffeeName;
    }
```

```java
    // 模板方法定义咖啡制作过程
    final void makeCoffee() {
        System.out.println("Making " + coffeeName + ":");
        grindCoffeeBeans();
        brewCoffee();
        addCondiments();
        System.out.println();
    }

    // 具体步骤的具体实现由子类提供
    abstract void grindCoffeeBeans();
    abstract void brewCoffee();

    // 添加调料的默认实现
    void addCondiments() {
        System.out.println("Adding condiments");
    }
}

// 具体的美式咖啡类
class AmericanCoffeeMaker extends CoffeeMakerTemplate {
    // 构造函数传递咖啡名称
    public AmericanCoffeeMaker() {
        super("American Coffee");
    }

    @Override
    void grindCoffeeBeans() {
        System.out.println("Grinding coffee beans");
    }

    @Override
    void brewCoffee() {
        System.out.println("Brewing coffee");
    }
}

// 具体的拿铁咖啡类
class LatteCoffeeMaker extends CoffeeMakerTemplate {
    // 构造函数传递咖啡名称
    public LatteCoffeeMaker() {
        super("Latte");
    }

    @Override
    void grindCoffeeBeans() {
        System.out.println("Grinding coffee beans");
    }
```

```java
    @Override
    void brewCoffee() {
        System.out.println("Brewing coffee");
    }

    // 添加调料的特定实现
    @Override
    void addCondiments() {
        System.out.println("Adding milk");
        System.out.println("Adding condiments");
    }
}

// 客户端代码
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        while (scanner.hasNext()) {
            int coffeeType = scanner.nextInt();

            CoffeeMakerTemplate coffeeMaker = null;

            if (coffeeType == 1) {
                coffeeMaker = new AmericanCoffeeMaker();
            } else if (coffeeType == 2) {
                coffeeMaker = new LatteCoffeeMaker();
            } else {
                System.out.println("Invalid coffee type");
                continue;
            }

            // 制作咖啡
            coffeeMaker.makeCoffee();
        }
    }
}
```

迭代器模式

题目链接

[迭代器模式-学生名单]

基本概念

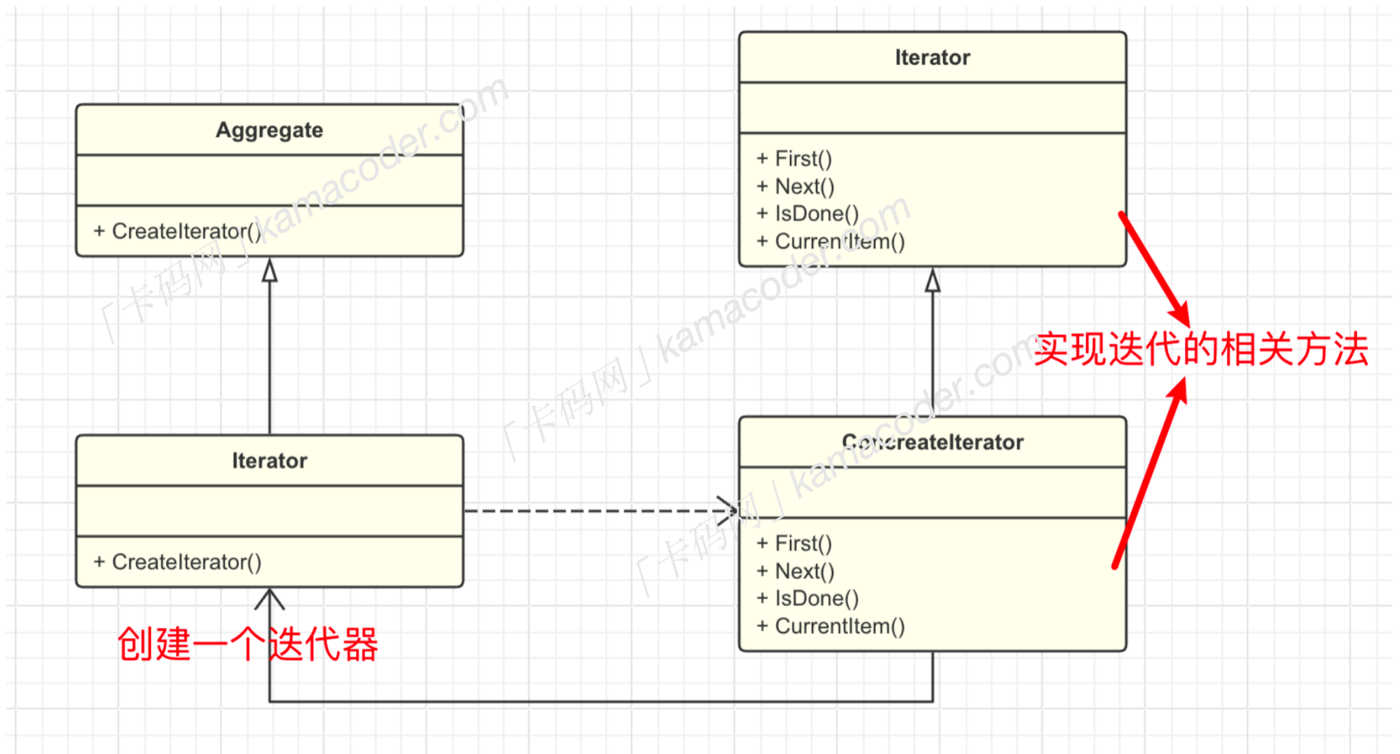
迭代器模式是一种行为设计模式，是一种使用频率非常高的设计模式，在各个语言中都有应用，其主要目的是**提供一种统一的方式来访问一个聚合对象中的各个元素，**而不需要暴露该对象的内部表示。通过迭代器，客户端可以顺序访问聚合对象的元素，而无需了解底层数据结构。

> 迭代器模式应用广泛，但是大多数语言都已经内置了迭代器接口，不需要自己实现。

基本结构

迭代器模式包括以下几个重要角色

- 迭代器接口`Iterator`：定义访问和遍历元素的接口，通常会包括`hasNext()`方法用于检查是否还有下一个元素，以及`next()`方法用于获取下一个元素。有的还会实现获取第一个元素以及获取当前元素的方法。
- 具体迭代器`ConcreateIterator`：实现迭代器接口，实现遍历逻辑对聚合对象进行遍历。
- 抽象聚合类：定义了创建迭代器的接口，包括一个`createIterator`方法用于创建一个迭代器对象。
- 具体聚合类：实现在抽象聚合类中声明的`createIterator()`方法，返回一个与具体聚合对应的具体迭代器



简易实现

1. 定义迭代器接口：通常会有检查是否还有下一个元素以及获取下一个元素的方法。

```java
// 迭代器接口
public interface Iterator{
    // 检查是否还会有下一个元素
    boolean hasNext();
    // 获取下一个元素
    Object next();
}
```

2. 定义具体迭代器：实现迭代器接口，遍历集合。

```
public class ConcreteIterator implements Iterator {
    private int index;
    private List<Object> elements;

    // 构造函数初始化迭代器
    public ConcreteIterator(List<Object> elements) {
        this.elements = elements;
        this.index = 0;
    }

    @Override
    public boolean hasNext() {
        return index < elements.size();
    }

    @Override
    public Object next() {
        if (hasNext()) {
            return elements.get(index++);
        }
        return null;
    }
}
```

3. 定义聚合接口：通常包括`createIterator()`方法，用于创建迭代器

```
public interface Iterable {
    Iterator createIterator();
}
```

4. 实现具体聚合：创建具体的迭代器

```
// 具体聚合
public class ConcreteIterable implements Iterable {
    private List<Object> elements;

    // 构造函数初始化可迭代对象
    public ConcreteIterable(List<Object> elements) {
        this.elements = elements;
    }

    @Override
    public Iterator createIterator() {
        return new ConcreteIterator(elements);
    }
}
```

5. 客户端使用

```java
import java.util.ArrayList;
import java.util.List;

public class IteratorPatternExample {
    public static void main(String[] args) {
        List<Object> elements = new ArrayList<>();
        elements.add("Element 1");
        elements.add("Element 2");
        elements.add("Element 3");

        Iterable iterable = new ConcreteIterable(elements);
        Iterator iterator = iterable.createIterator();

        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

使用场景

迭代器模式是一种通用的设计模式，其封装性强，简化了客户端代码，客户端不需要知道集合的内部结构，只需要关心迭代器和迭代接口就可以完成元素的访问。但是引入迭代器模式会增加额外的类，每增加一个集合类，都需要增加该集合对应的迭代器，这也会使得代码结构变得更加复杂。

许多编程语言和框架都使用了这个模式提供一致的遍历和访问集合元素的机制。下面是几种常见语言迭代器模式的实现。

1. Java语言：集合类（如ArrayList、LinkedList), 通过`Iterator`接口，可以遍历集合中的元素。

```java
List<String> list = new ArrayList<>();
list.add("Item 1");
list.add("Item 2");
list.add("Item 3");

Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

2. Python语言：使用迭代器和生成器来实现迭代模式，`iter()`和`next()`函数可以用于创建和访问迭代器。

```python
elements = ["Element 1", "Element 2", "Element 3"]
iterator = iter(elements)

while True:
    try:
        element = next(iterator)
        print(element)
    except StopIteration:
        break
```

3. C++语言：C++中的STL提供了迭代器的支持，`begin()`和`end()`函数可以用于获取容器的起始和结束迭代器。

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<std::string> elements = {"Element 1", "Element 2", "Element 3"};

    for (auto it = elements.begin(); it != elements.end(); ++it) {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

4. JavaScript语言：ES6中新增了迭代器协议，使得遍历和访问集合元素变得更加方便。

```javascript
// 可迭代对象实现可迭代协议
class IterableObject {
  constructor() {
    this.elements = [];
  }
  addElement(element) {
    this.elements.push(element);
  }
  [Symbol.iterator]() {
    let index = 0;
    // 迭代器对象实现迭代器协议
    return {
      next: () => {
        if (index < this.elements.length) {
          return { value: this.elements[index++], done: false };
        } else {
          return { done: true };
        }
      }
    };
```

```
  }
}

// 使用迭代器遍历可迭代对象
const iterableObject = new IterableObject();
iterableObject.addElement("Element 1");
iterableObject.addElement("Element 2");
iterableObject.addElement("Element 3");

for (const element of iterableObject) {
  console.log(element);
}
```

本题代码

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

// 可迭代对象接口
interface StudentCollection {
    java.util.Iterator<Student> iterator();
}

// 具体可迭代对象
class ConcreteStudentCollection implements StudentCollection {
    private List<Student> students = new ArrayList<>();

    public void addStudent(Student student) {
        students.add(student);
    }

    @Override
    public java.util.Iterator<Student> iterator() {
        return new ConcreteStudentIterator(students);
    }
}

// 迭代器接口
interface Iterator<T> {
    boolean hasNext();

    T next();
}

// 具体迭代器
class ConcreteStudentIterator implements java.util.Iterator<Student> {
    private List<Student> students;
```

```java
    private int currentIndex = 0;

    public ConcreteStudentIterator(List<Student> students) {
        this.students = students;
    }

    @Override
    public boolean hasNext() {
        return currentIndex < students.size();
    }

    @Override
    public Student next() {
        if (hasNext()) {
            Student student = students.get(currentIndex);
            currentIndex++;
            return student;
        }
        return null;
    }
}

// 学生类
class Student {
    private String name;
    private String studentId;

    public Student(String name, String studentId) {
        this.name = name;
        this.studentId = studentId;
    }

    public String getName() {
        return name;
    }

    public String getStudentId() {
        return studentId;
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取学生数量
        int n = scanner.nextInt();
        scanner.nextLine(); // 读取换行符
```

```
        // 创建具体可迭代对象
        ConcreteStudentCollection studentCollection = new ConcreteStudentCollection();

        // 读取学生信息并添加到集合
        for (int i = 0; i < n; i++) {
            String[] input = scanner.nextLine().split(" ");
            if (input.length == 2) {
                String name = input[0];
                String studentId = input[1];
                Student student = new Student(name, studentId);
                studentCollection.addStudent(student);
            } else {
                System.out.println("Invalid input");
                return;
            }
        }

        // 使用迭代器遍历学生集合
        java.util.Iterator<Student> iterator = studentCollection.iterator();
        while (iterator.hasNext()) {
            Student student = iterator.next();
            System.out.println(student.getName() + " " + student.getStudentId());
        }
    }
}
```

状态模式

题目链接

[状态模式-开关台灯]

基本结构

状态模式（State Pattern）是一种行为型设计模式，它适用于一个对象在在不同的状态下有不同的行为时，比如说电灯的开、关、闪烁是不停的状态，状态不同时，对应的行为也不同，在没有状态模式的情况下，为了添加新的状态或修改现有的状态，往往**需要修改已有的代码**，这违背了开闭原则，而且如果对象的状态切换逻辑和各个状态的行为都在同一个类中实现，就可能导致该类的职责过重，不符合单一职责原则。

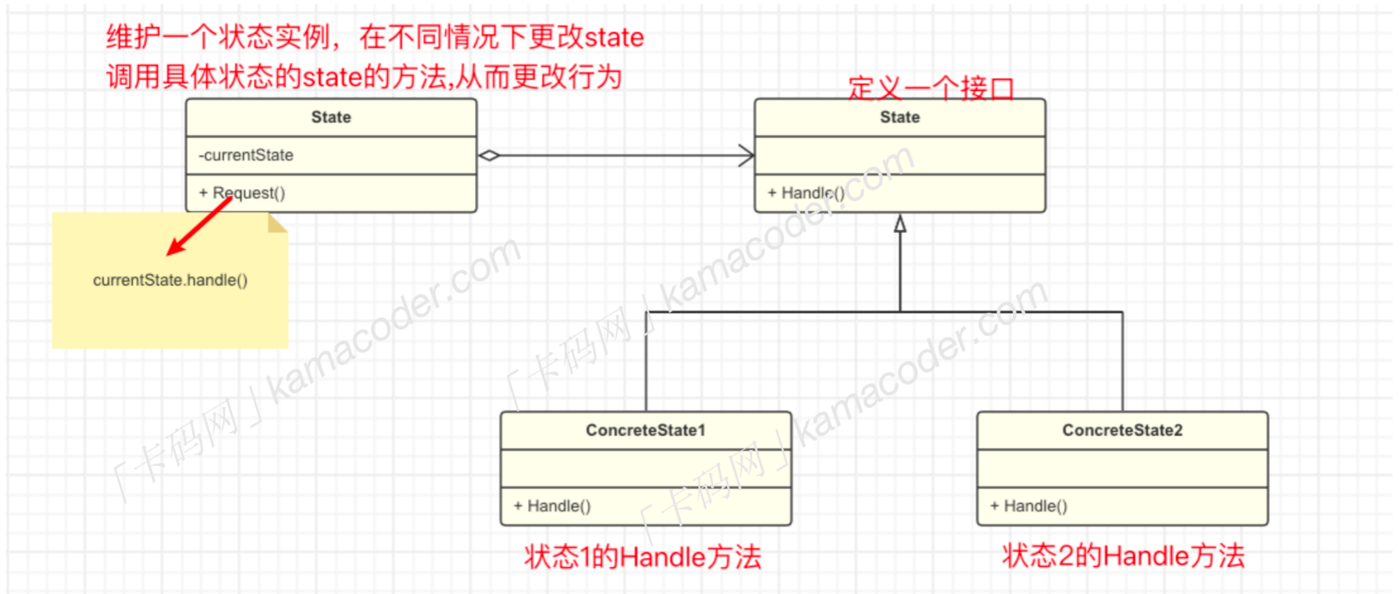
而状态模式将每个状态的行为封装在一个具体状态类中，使得每个状态类相对独立，并将对象在不同状态下的行为进行委托，从而使得对象的状态可以在运行时动态改变，每个状态的实现也不会影响其他状态。

基本结构

状态模式包括以下几个重要角色：

- `State`（状态）：定义一个接口，用于封装与Context的一个特定状态相关的行为。
- `ConcreteState`（具体状态）：负责处理Context在状态改变时的行为, 每一个具体状态子类实现一个与`Context`的一个状态相关的行为。

- `Context`（上下文）：维护一个具体状态子类的实例，这个实例定义当前的状态。



基本使用

1. 定义状态接口：创建一个状态接口，该接口声明了对象可能的各种状态对应的方法。

```
// 状态接口
public interface State {
    void handle();
}
```

2. 实现具体状态类： 为对象可能的每种状态创建具体的状态类，实现状态接口中定义的方法。

```
// 具体状态类1
public class ConcreteState1 implements State {
    @Override
    public void handle() {
        // 执行在状态1下的操作
    }
}

// 具体状态类2
public class ConcreteState2 implements State {
    @Override
    public void handle() {
        // 执行在状态2下的操作
    }
}
```

3. 创建上下文类：该类包含对状态的引用，并在需要时调用当前状态的方法。

```java
// 上下文类
public class Context {
    private State currentState;

    public void setState(State state) {
        this.currentState = state;
    }

    public void request() {
        currentState.handle();
    }
}
```

4. 客户端使用：创建具体的状态对象和上下文对象，并通过上下文对象调用相应的方法。通过改变状态，可以改变上下文对象的行为

```java
public class Client {
    public static void main(String[] args) {
        Context context = new Context();

        State state1 = new ConcreteState1();
        State state2 = new ConcreteState2();

        context.setState(state1);
        context.request(); // 执行在状态1下的操作

        context.setState(state2);
        context.request(); // 执行在状态2下的操作
    }
}
```

使用场景

状态模式将每个状态的实现都封装在一个类中，每个状态类的实现相对独立，使得添加新状态或修改现有状态变得更加容易，避免了使用大量的条件语句来控制对象的行为。但是如果状态过多，会导致类的数量增加，可能会使得代码结构复杂。

总的来说，状态模式适用于有限状态机的场景，其中对象的行为在运行时可以根据内部状态的改变而改变，在游戏开发中，Unity3D 的 Animator 控制器就是一个状态机。它允许开发人员定义不同的状态（动画状态），并通过状态转换来实现角色的动画控制和行为切换。

本题代码

```java
import java.util.Scanner;

// 状态接口
interface State {
```

```
    String handle(); // // 处理状态的方法
}

// 具体状态类
class OnState implements State {
    @Override
    public String handle() {
        return "Light is ON";
    }
}

class OffState implements State {
    @Override
    public String handle() {
        return "Light is OFF";
    }
}

class BlinkState implements State {
    @Override
    public String handle() {
        return "Light is Blinking";
    }
}

// 上下文类
class Light {
    private State state; // 当前状态

    public Light() {
        this.state = new OffState(); // 初始状态为关闭
    }

    public void setState(State state) { // 设置新的状态
        this.state = state;
    }

    public String performOperation() { // 执行当前状态的操作
        return state.handle();
    }
}

public class Main {
    public static void main(String[] args) {
        // 创建一个Scanner对象以读取用户输入
        Scanner scanner = new Scanner(System.in);

        int n = scanner.nextInt();
        scanner.nextLine();
```

```java
        Light light = new Light();
    // 处理用户输入
        for (int i = 0; i < n; i++) {
            String command = scanner.nextLine().trim();
      // 根据输入修改灯的状态
            switch (command) {
                case "ON":
                    light.setState(new OnState());
                    break;
                case "OFF":
                    light.setState(new OffState());
                    break;
                case "BLINK":
                    light.setState(new BlinkState());
                    break;
                default:
                    System.out.println("Invalid command: " + command);
                    break;
            }
      // 显示灯的当前状态
            System.out.println(light.performOperation());
        }
    }
}
```

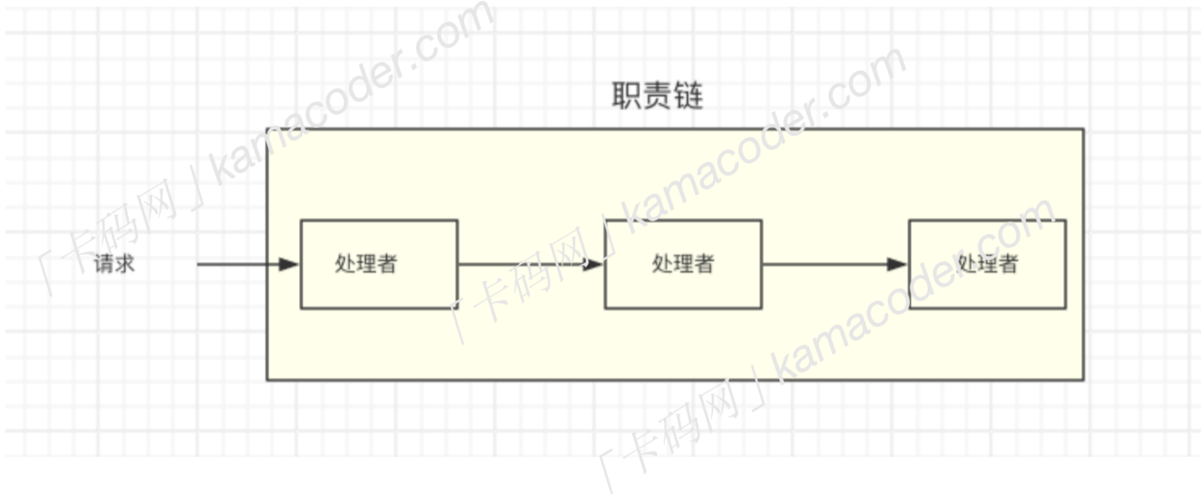
责任链模式

题目链接

[责任链模式-请假审批]

基本概念

责任链模式是一种行为型设计模式，它允许你构建一个对象链，让请求从链的一端进入，然后沿着链上的对象依次处理，直到链上的某个对象能够处理该请求为止。

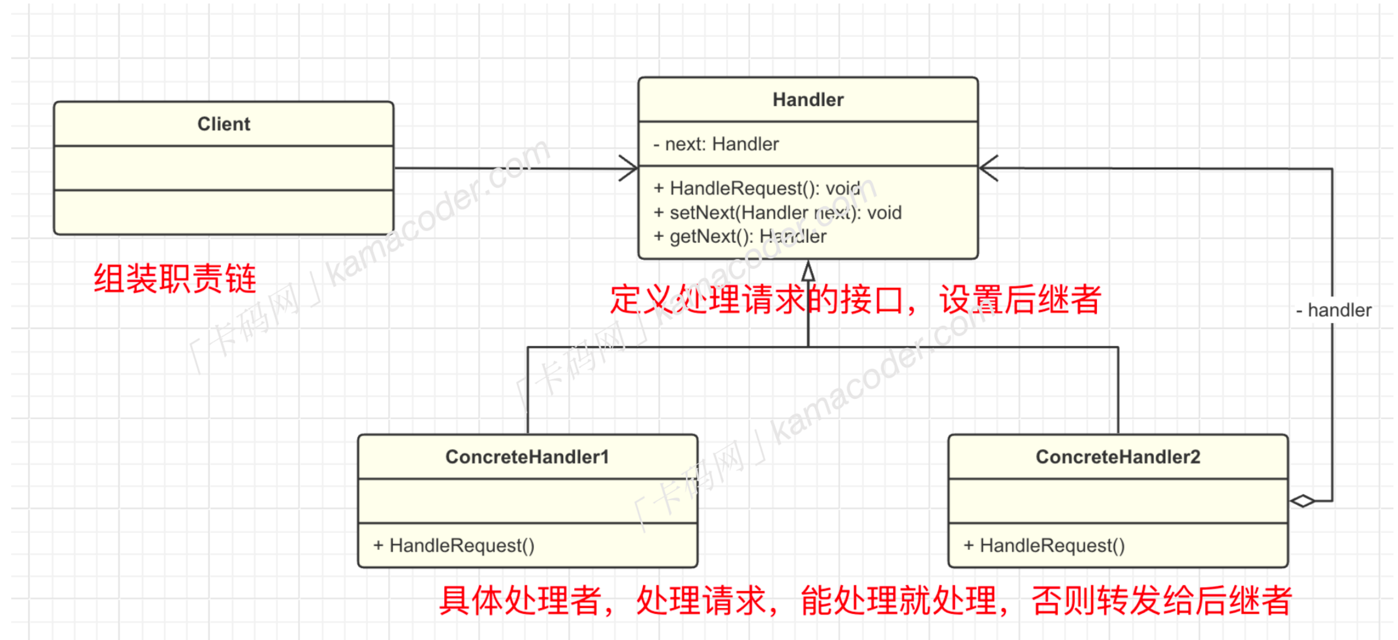


职责链上的处理者就是一个对象，可以对请求进行处理或者将请求转发给下一个节点，这个场景在生活中很常见，就是一个逐层向上递交的过程，最终的请求要么被处理者所处理，要么处理不了，这也因此可能导致请求无法被处理。

组成结构

责任链模式包括以下几个基本结构：

1. 处理者`Handler`：定义一个处理请求的接口，包含一个处理请求的抽象方法和一个指向下一个处理者的链接。
2. 具体处理者`ConcreteHandler`: 实现处理请求的方法，并判断能否处理请求，如果能够处理请求则进行处理，否则将请求传递给下一个处理者。
3. 客户端：创建并组装处理者对象链，并将请求发送到链上的第一个处理者。



简易实现

1. 处理者：定义处理请求的接口

```
interface Handler {
    // 处理请求的方法
    void handleRequest(double amount);
    // 设置下一个处理者的方法
    void setNextHandler(Handler nextHandler);
}
```

2. 具体处理者：实现处理请求

```
class ConcreteHandler implements Handler {
    private Handler nextHandler;

    @Override
    public void handleRequest(Request request) {
        // 根据具体情况处理请求，如果无法处理则交给下一个处理者
        if (canHandle(request)) {
            // 处理请求的逻辑
        } else if (nextHandler != null) {
            // 交给下一个处理者处理
            nextHandler.handleRequest(request);
        } else {
            // 无法处理请求的逻辑
        }
    }

    @Override
    public void setNextHandler(Handler nextHandler) {
        this.nextHandler = nextHandler;
    }

    // 具体处理者自己的判断条件
    private boolean canHandle(Request request) {
        // 根据具体情况判断是否能够处理请求
        return /* 判断条件 */;
    }
}
```

3. 客户端创建并组装处理者对象链，将请求发送给链上第一个处理者

```
public class Main {
    public static void main(String[] args) {
        // 创建处理者实例
        Handler handler1 = new ConcreteHandler();
        Handler handler2 = new ConcreteHandler();
```

```java
        // ...

        // 构建责任链
        handler1.setNextHandler(handler2);
        // ...

        // 发送请求
        Request request = new Request(/* 请求参数 */);
        handler1.handleRequest(request);
    }
}
```

使用场景

责任链模式具有下面几个优点：

- 降低耦合度：将请求的发送者和接收者解耦，每个具体处理者都只负责处理与自己相关的请求，客户端不需要知道具体是哪个处理者处理请求。
- 增强灵活性：可以动态地添加或删除处理者，改变处理者之间的顺序以满足不同需求。

但是由于一个请求可能会经过多个处理者，这可能会导致一些性能问题，并且如果整个链上也没有合适的处理者来处理请求，就会导致请求无法被处理。

责任链模式是设计模式中简单且常见的设计模式，在日常中也会经常使用到，比如Java开发中过滤器的链式处理，以及Spring框架中的拦截器，都组装成一个处理链对请求、响应进行处理。

本题代码

```java
import java.util.Scanner;

// 处理者：定义接口
interface LeaveHandler {
    void handleRequest(LeaveRequest request);
}

// 具体处理者：可以有多个，负责具体处理，这里分为 Supervisor、Manager、Director
class Supervisor implements LeaveHandler {
    private static final int MAX_DAYS_SUPERVISOR_CAN_APPROVE = 3;
    private LeaveHandler nextHandler;

    public Supervisor(LeaveHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    @Override
    public void handleRequest(LeaveRequest request) {
        if (request.getDays() <= MAX_DAYS_SUPERVISOR_CAN_APPROVE) {
            System.out.println(request.getName() + " Approved by Supervisor.");
        } else if (nextHandler != null) {
```

```java
            nextHandler.handleRequest(request);
        } else {
            System.out.println(request.getName() + " Denied by Supervisor.");
        }
    }
}

class Manager implements LeaveHandler {
    private static final int MAX_DAYS_MANAGER_CAN_APPROVE = 7;
    private LeaveHandler nextHandler;

    public Manager(LeaveHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    @Override
    public void handleRequest(LeaveRequest request) {
        if (request.getDays() <= MAX_DAYS_MANAGER_CAN_APPROVE) {
            System.out.println(request.getName() + " Approved by Manager.");
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        } else {
            System.out.println(request.getName() + " Denied by Manager.");
        }
    }
}

class Director implements LeaveHandler {
    private static final int MAX_DAYS_DIRECTOR_CAN_APPROVE = 10;

    @Override
    public void handleRequest(LeaveRequest request) {
        if (request.getDays() <= MAX_DAYS_DIRECTOR_CAN_APPROVE) {
            System.out.println(request.getName() + " Approved by Director.");
        } else {
            System.out.println(request.getName() + " Denied by Director.");
        }
    }
}

// 请求类
class LeaveRequest {
    private String name;
    private int days;

    public LeaveRequest(String name, int days) {
        this.name = name;
        this.days = days;
    }
```

```
    public String getName() {
        return name;
    }

    public int getDays() {
        return days;
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int n = scanner.nextInt();
        scanner.nextLine();
    // 组装职责链
        LeaveHandler director = new Director();
        LeaveHandler manager = new Manager(director);
        LeaveHandler supervisor = new Supervisor(manager);

        for (int i = 0; i < n; i++) {
            String[] input = scanner.nextLine().split(" ");
            if (input.length == 2) {
                String name = input[0];
                int days = Integer.parseInt(input[1]);
                LeaveRequest request = new LeaveRequest(name, days);
                supervisor.handleRequest(request);
            } else {
                System.out.println("Invalid input");
                return;
            }
        }
    }
}
```

解释器模式

题目链接

解释器模式-数学表达式

基本概念

解释器模式（Interpreter Pattern）是一种行为型设计模式，它定义了一个语言的文法，并且建立一个【解释器】来解释该语言中的句子。

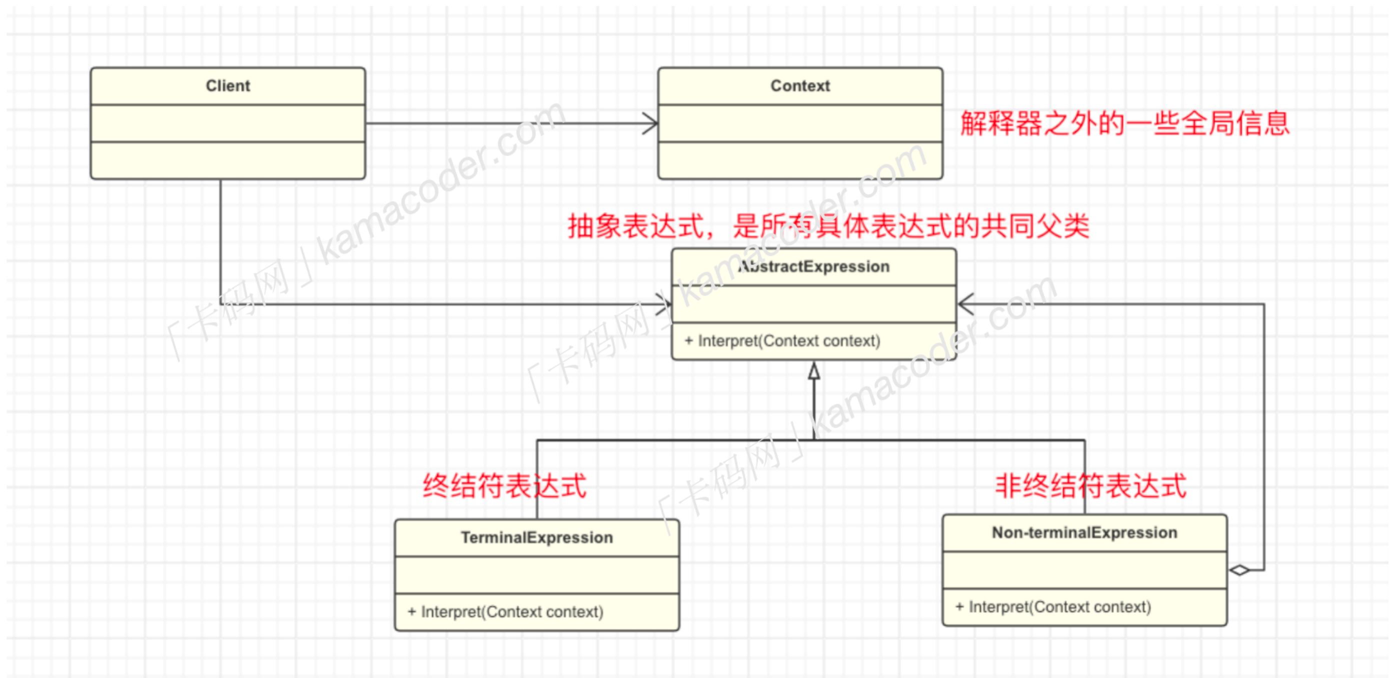
比如说SQL语法、正则表达式，这些内容比较简短，但是表达的内容可不仅仅是字面上的那些符号，计算机想要理解这些语法，就需要解释这个语法规则，因此解释器模式常用于实现编程语言解释器、正则表达式处理等场景。

组成结构

解释器模式主要包含以下几个角色：

1. **抽象表达式（Abstract Expression）**： 定义了解释器的接口，包含了解释器的方法 `interpret`。
2. **终结符表达式（Terminal Expression）**： 在语法中不能再分解为更小单元的符号。
3. **非终结符表达式（Non-terminal Expression）**： 文法中的复杂表达式，它由终结符和其他非终结符组成。
4. **上下文（Context）**： 包含解释器之外的一些全局信息，可以存储解释器中间结果，也可以用于向解释器传递信息。

> 举例来说，表达式 "3 + 5 * 2"，数字 "3" 和 "5"， "2" 是终结符，而运算符 "+", "*"都需要两个操作数，属于非终结符。



简易实现

1. 创建抽象表达式接口： 定义解释器的接口，声明一个 `interpret` 方法，用于解释语言中的表达式。

```
// 抽象表达式接口
public interface Expression {
    int interpret();
}
```

2. 创建具体的表达式类： 实现抽象表达式接口，用于表示语言中的具体表达式。

```java
public class TerminalExpression implements Expression {
    private int value;

    public TerminalExpression(int value) {
        this.value = value;
    }

    @Override
    public int interpret() {
        return value;
    }
}
```

3. 非终结符表达式：抽象表达式的一种，用于表示语言中的非终结符表达式，通常包含其他表达式。

```java
public class AddExpression implements Expression {
    private Expression left;
    private Expression right;

    public AddExpression(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public int interpret() {
        return left.interpret() + right.interpret();
    }
}
```

4. 上下文：包含解释器需要的一些全局信息或状态。

```java
public class Context {
    // 可以在上下文中存储一些全局信息或状态
}
```

5. 客户端：构建并组合表达式，然后解释表达式。

```java
public class Main {
    public static void main(String[] args) {
        Context context = new Context();

        Expression expression = new AddExpression(
                new TerminalExpression(1),
                new TerminalExpression(2)
        );
```

```
        int result = expression.interpret();
        System.out.println("Result: " + result);
    }
}
```

使用场景

当需要解释和执行特定领域或业务规则的语言时，可以使用解释器模式。例如，SQL解释器、正则表达式解释器等。但是需要注意的是解释器模式可能会导致类的层次结构较为复杂，同时也可能不够灵活，使用要慎重。

本题代码

```java
import java.util.Scanner;
import java.util.Stack;

// 抽象表达式接口
interface Expression {
    int interpret();
}

// 终结符表达式类 - 数字
class NumberExpression implements Expression {
    private int number;

    public NumberExpression(int number) {
        this.number = number;
    }

    @Override
    public int interpret() {
        return number;
    }
}

// 非终结符表达式类 - 加法
class AddExpression implements Expression {
    private Expression left;
    private Expression right;

    public AddExpression(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public int interpret() {
```

```java
        return left.interpret() + right.interpret();
    }
}

// 非终结符表达式类 - 乘法
class MultiplyExpression implements Expression {
    private Expression left;
    private Expression right;

    public MultiplyExpression(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public int interpret() {
        return left.interpret() * right.interpret();
    }
}

// 上下文类
class Context {
    private Stack<Expression> expressionStack = new Stack<>();

    public void pushExpression(Expression expression) {
        expressionStack.push(expression);
    }

    public Expression popExpression() {
        return expressionStack.pop();
    }
}

public class Main{
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Context context = new Context();

        // 处理用户输入的数学表达式
        while (scanner.hasNextLine()) {
            String userInput = scanner.nextLine();
            Expression expression = parseExpression(userInput);
            if (expression != null) {
                context.pushExpression(expression);
                System.out.println(expression.interpret());
            } else {
                System.out.println("Invalid expression.");
            }
        }
```

```
        scanner.close();
    }

    // 解析用户输入的数学表达式并返回相应的抽象表达式类
    private static Expression parseExpression(String userInput) {
        try {
            Stack<Expression> expressionStack = new Stack<>();
            char[] tokens = userInput.toCharArray();

            for (int i = 0; i < tokens.length; i++) {
                char token = tokens[i];

                if (Character.isDigit(token)) {
                    expressionStack.push(new
NumberExpression(Character.getNumericValue(token)));

                    // 如果下一个字符不是数字，且栈中有两个以上的元素，说明可以进行运算
                    if (i + 1 < tokens.length && !Character.isDigit(tokens[i + 1]) &&
expressionStack.size() >= 2) {
                        Expression right = expressionStack.pop();
                        Expression left = expressionStack.pop();
                        char operator = tokens[i + 1];

                        if (operator == '+') {
                            expressionStack.push(new AddExpression(left, right));
                        } else if (operator == '*') {
                            expressionStack.push(new MultiplyExpression(left, right));
                        }

                        i++; // 跳过下一个字符，因为已经处理过了
                    }
                } else {
                    return null; // Invalid token
                }
            }

            return expressionStack.pop();
        } catch (Exception e) {
            return null;
        }
    }
}
```

访问者模式

题目链接

[访问者模式-图形的面积]

基本概念

访问者模式（Visitor Pattern）是一种行为型设计模式，可以在不改变对象结构的前提下，对对象中的元素进行新的操作。

举个例子，假设有一个动物园，里面有不同种类的动物，比如狮子、大象、猴子等。每个动物都会被医生检查身体，被管理员投喂，被游客观看，医生，游客，管理员都属于访问者。

```
// 定义动物接口
interface Animal {
    void accept(Visitor visitor);
}

// 具体元素类：狮子
class Lion implements Element {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

// 具体元素类：大象
class Elephant implements Element {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

// 具体元素类：猴子
class Monkey implements Element {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
```

如果你想对动物园中的每个动物执行一些操作，比如医生健康检查、管理员喂食、游客观赏等。就可以使用访问者模式来实现这些操作。

```
// 定义访问者接口
interface Visitor {
    void visit(Animal animal);
}
```

```java
// 具体访问者类：医生
class Vet implements Visitor {
    @Override
    public void visit(Animal animal) {

    }
}

// 具体访问者类：管理员
class Zookeeper implements Visitor {
    @Override
    public void visit(Animal animal) {
    }
}

// 具体访问者类：游客
class VisitorPerson implements Visitor {
    @Override
    public void visit(Animal animal) {
    }
}
```

将这些访问者应用到动物园的每个动物上

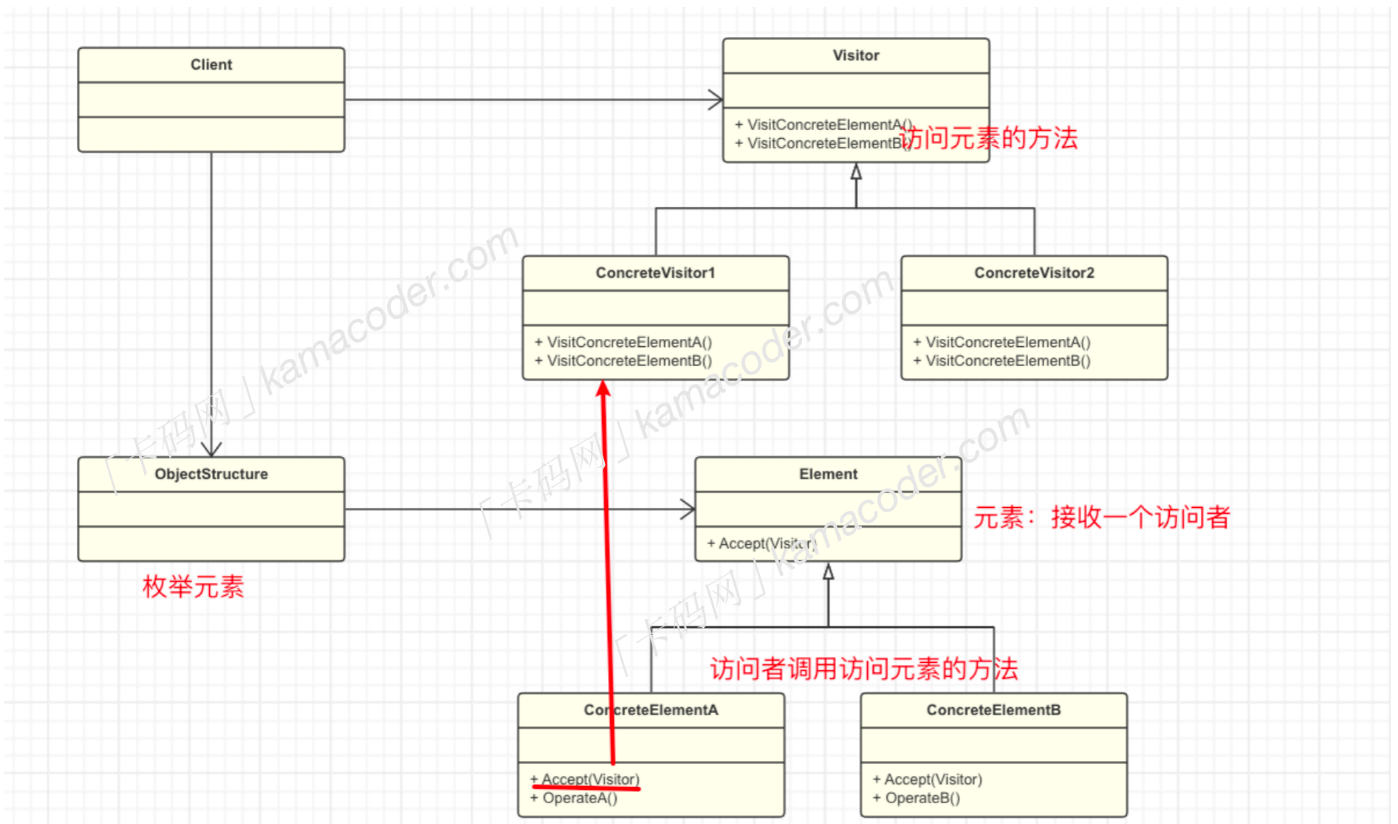
```java
public class Main {
    public static void main(String[] args) {
        Animal lion = new Lion();
        Animal elephant = new Elephant();
        Animal monkey = new Monkey();

        Visitor vet = new Vet();
        Visitor zookeeper = new Zookeeper();
        Visitor visitorPerson = new VisitorPerson();

        // 动物接受访问者的访问
        lion.accept(vet);
        elephant.accept(zookeeper);
        monkey.accept(visitorPerson);
    }
}
```

基本结构

访问者模式包括以下几个基本角色：

- **抽象访问者（Visitor）**： 声明了访问者可以访问哪些元素，以及如何访问它们的方法`visit`。
- **具体访问者（ConcreteVisitor）**： 实现了抽象访问者定义的方法，不同的元素类型可能有不同的访问行为。医生、管理员、游客都属于具体的访问者，它们的访问行为不同。
- **抽象元素（Element）**： 定义了一个accept方法，用于接受访问者的访问。

- **具体元素（ConcreteElement）**： 实现了accept方法，是访问者访问的目标。
- **对象结构（Object Structure）**： 包含元素的集合，可以是一个列表、一个集合或者其他数据结构。负责遍历元素，并调用元素的接受方法。



简易实现：

1. 定义抽象访问者：声明那些元素可以访问

```
// 抽象访问者
interface Visitor {
    void visit(ConcreteElementA element);
    void visit(ConcreteElementB element);
}
```

2. 实现具体访问者：实现具体的访问逻辑

```
// 具体访问者A
class ConcreteVisitorA implements Visitor {
    @Override
    public void visit(ConcreteElementA element) {
        System.out.println("ConcreteVisitorA Visit ConcreteElementA");
    }

    @Override
    public void visit(ConcreteElementB element) {
        System.out.println("ConcreteVisitorA Visit ConcreteElementB");
```

```
    }
}

// 具体访问者B
class ConcreteVisitorB implements Visitor {
    @Override
    public void visit(ConcreteElementA element) {
        System.out.println("ConcreteVisitorB Visit ConcreteElementA");
    }

    @Override
    public void visit(ConcreteElementB element) {
        System.out.println("ConcreteVisitorB Visit ConcreteElementB");
    }
}
```

3. 定义元素接口：声明接收访问者的方法。

```
// 抽象元素
interface Element {
    void accept(Visitor visitor);
}
```

4. 实现具体元素：实现接受访问者的方法

```
// 具体元素A
class ConcreteElementA implements Element {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

// 具体元素B
class ConcreteElementB implements Element {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
```

5. 创建对象结构：提供一个接口让访问者访问它的元素。

```
// 对象结构
class ObjectStructure {
    private List<Element> elements = new ArrayList<>();

    public void attach(Element element) {
```

```
        elements.add(element);
    }

    public void detach(Element element) {
        elements.remove(element);
    }

    public void accept(Visitor visitor) {
        for (Element element : elements) {
            element.accept(visitor);
        }
    }
}
```

6. 客户端调用

```
public class Main {
    public static void main(String[] args) {
        ObjectStructure objectStructure = new ObjectStructure();
        objectStructure.attach(new ConcreteElementA());
        objectStructure.attach(new ConcreteElementB());

        Visitor visitorA = new ConcreteVisitorA();
        Visitor visitorB = new ConcreteVisitorB();

        objectStructure.accept(visitorA);
        objectStructure.accept(visitorB);
    }
}
```

使用场景

访问者模式结构较为复杂，但是访问者模式将同一类操作封装在一个访问者中，使得相关的操作彼此集中，提高了代码的可读性和维护性。它常用于**对象结构比较稳定，但经常需要在此对象结构上定义新的操作**，这样就无需修改现有的元素类，只需要定义新的访问者来添加新的操作。

本题代码

```
import java.util.Scanner;

// 元素接口
interface Shape {
    void accept(Visitor visitor);
}

// 具体元素类
class Circle implements Shape {
    private int radius;
```

```java
    public Circle(int radius) {
        this.radius = radius;
    }

    public int getRadius() {
        return radius;
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

class Rectangle implements Shape {
    private int width;
    private int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

// 访问者接口
interface Visitor {
    void visit(Circle circle);

    void visit(Rectangle rectangle);
}


// 具体访问者类
class AreaCalculator implements Visitor {
    @Override
```

```java
    public void visit(Circle circle) {
        double area = 3.14 * Math.pow(circle.getRadius(), 2);
        System.out.println(area);
    }

    @Override
    public void visit(Rectangle rectangle) {
        int area = rectangle.getWidth() * rectangle.getHeight();
        System.out.println(area);
    }
}

// 对象结构类
class Drawing {
    private Shape[] shapes;

    public Drawing(Shape[] shapes) {
        this.shapes = shapes;
    }

    public void accept(Visitor visitor) {
        for (Shape shape : shapes) {
            shape.accept(visitor);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int n = scanner.nextInt();
        scanner.nextLine();
    // 创建一个数组来存储图形对象
        Shape[] shapes = new Shape[n];
    // 根据用户输入创建不同类型的图形对象
        for (int i = 0; i < n; i++) {
            String[] input = scanner.nextLine().split(" ");
            if (input[0].equals("Circle")) {
                int radius = Integer.parseInt(input[1]);
                shapes[i] = new Circle(radius);
            } else if (input[0].equals("Rectangle")) {
                int width = Integer.parseInt(input[1]);
                int height = Integer.parseInt(input[2]);
                shapes[i] = new Rectangle(width, height);
            } else {
                System.out.println("Invalid input");
                return;
            }
```

```
        }
    // 创建一个图形集合
        Drawing drawing = new Drawing(shapes);
        // 创建一个面积计算访问者
        Visitor areaCalculator = new AreaCalculator();
        // 访问图形集合并计算面积
        drawing.accept(areaCalculator);
    }
}
```